BRAINformat: A Data Standardization Framework for Neuroscience Data

Oliver Rübel [1,2,*], Prabhat [2], Peter Denes [3], David Conant [4], Edward Chang [4], and Kristofer Bouchard [5]

[1]Computational Research Division, Lawrence Berkeley National Laboratory (LBNL), Berkeley, CA, USA
[2]National Energy Research Scientific Computing Center, LBNL, Berkeley, CA, USA
[3]Physical Sciences Division, LBNL, Berkeley, CA, USA
[4]UCSF Medical Center, University of California San Francisco, San Francisco, CA, USA
[5]Biological Systems and Engineering Division, LBNL, CA, USA

August, 2015

Acknowledgment

This work was supported by Laboratory Directed Research and Development (LDRD) funding from Berkeley Lab, provided by the Director, Office of Science, of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231. This research used resources of the National Energy Research Scientific Computing Center, a DOE Office of Science User Facility supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231.

Legal Disclaimer

BRAINformat: A Data Standardization Framework for Neuroscience Data

Oliver Rübel [1,2,*], **Prabhat** [2], **Peter Denes** [3], **David Conant** [4], **Edward Chang** [4], **and Kristofer Bouchard** [5]

[1] *Computational Research Division, Lawrence Berkeley National Laboratory (LBNL), Berkeley, CA, USA*
[2] *National Energy Research Scientific Computing Center, LBNL, Berkeley, CA, USA*
[3] *Physical Sciences Division, LBNL, Berkeley, CA, USA*
[4] *UCSF Medical Center, University of California San Francisco, San Francisco, CA, USA*
[5] *Biological Systems and Engineering Division, LBNL, CA, USA*

Correspondence*:
Oliver Rübel
Computational Research Division, Lawrence Berkeley National Laboratory, 1 Cyclotron Road, M/S 50F1650, Berkeley, CA, 94720, USA, oruebel@lbl.gov

ABSTRACT

Neuroscience is entering the era of 'extreme data' with little experience and few plans for the associated volume, velocity, variety, and veracity challenges. This is a serious impediment for both the sharing of data across labs, as well as the utilization of modern and high-performance computing capabilities to enable data driven discovery. Here, we introduce BRAINformat, a novel file format and model for management and storage of neuroscience data. The BRAINformat library defines application-independent design concepts and modules that together create a general framework for standardization of scientific data.

We describe the formal specification of scientific data standards, which facilitates sharing and verification of data and formats. We introduce the concept of *Managed Objects*, enabling semantic components of data formats to be specified as self-contained units, supporting modular and reusable design of data format components and file storage. The BRAINformat is built off of HDF5, enabling portable, scalable, and self-describing data storage. We introduce the novel concept of *Relationship Attributes* for modeling and use of semantic relationships between data objects, and discuss the annotation of data using dedicated data annotation modules provided by the BRAINformat library. Based on these concepts we implement dedicated, application-oriented modules and design a data standard for neuroscience data.The BRAINformat software library is open source, easy-to-use, and provides detailed user and developer documentation and is freely available at: `https://bitbucket.org/oruebel/brainformat`.

Keywords: data format specification, neuroscience data format

1 INTRODUCTION

Neuroscience research is facing an increasingly challenging 'big data' problem due to the growing complexity of experiments and the volume/variety of data being collected from many acquisition modalities. Neuroscientists are routinely collecting data in a broad range of data formats that are often highly domain specific, ad-hoc and/or designed for efficiency with respect to very specific tools and data types. Even for single experiments, scientists are interacting with often tens of different formats—one for each recording device and/or analysis—while many data standards are not well-described or are only accessible via proprietary software. Navigating this quagmire of formats hinders efficient data analysis, data sharing, and collaboration and can lead to errors and misinterpretation of data. File formats and data standards that can represent complex neuroscience data and make the data easily accessible play a key role in enabling scientific discovery, development of reusable tools for data analytics, and progress towards fostering collaboration in the neuroscience community.

The requirements towards a data format standard for neuroscience are highly complex and go far beyond the needs of traditional, data modality-specific formats (e.g. image, audio, or video formats). A neuroscience data format needs to support the management and organization of complex collections of data from many modalities and sources, e.g., neurological recordings, audio and video recordings, eye-tracking, motion tracking, task contingencies, external stimuli, derived analytic results, and many others. To enable data interpretation and analysis, the format needs to also support storage of complex metadata, e.g., descriptions of recording devices, experiments, subjects etc..

Advanced neurosciences analytics furthermore rely on complex data access patterns driven by data semantics. For example, to study human brain activity underlying speech, scientists need to be able to efficiently annotate and extract data using complex combinations of annotations. Annotating data in itself, however, is a highly complex task that requires the coordinated access to related data sources. For example, a scientists may use audio or video recordings to identify particular events of interest and in turn needs to locate the corresponding data in a neural recording dataset to annotate it. Therefore, it is crucial that neuroscience formats support annotation of data as well as the specification and use of relationships between data objects.

In addition to these more application-specific needs, a usable, sustainable, and extensible data format also needs to satisfy a broad range of general, advanced file format and API requirements — e.g, the format should be self-describing, easy-to-use, efficient, portable, scalable, verifiable, easy to share and should support self-contained and modular storage of large data. Meeting all these complex needs is a daunting challenge. Arguably, the focus of a neuroscience-oriented data standard should be on addressing the application-centric needs of organizing scientific data and metadata, rather than on reinventing file storage and format methods. For the development of BRAINformat we have utilized HDF5 as the basic storage format as it already satisfies a broad range of the more basic format requirements—HDF5 is self-describing, portable, extensible, widely supported by programming languages and analysis tools, and is optimized for storage and I/O of large-scale scientific data.

In this manuscript we introduce the BRAINformat, a novel data format standardization framework and API for scientific data, developed at the Lawrence Berkeley National Labs in collaboration with neuroscientists at UCB and UCSF. BRAINformat supports the formal specification and verification of scientific data formats and supports the organization of data in a modular, extensible, and reusable fashion via the concept of *managed objects* (Sec. 3.1). To enable the modeling and use of complex relationships between data objects, we introduce the novel concept or *relationship attributes*. Relationship attributes support the specification of structural and semantic links between data, enabling users and developers to formally document and utilize relationships in a well-structured and programmatic fashion (Sec. 3.2). We demonstrate the use of chains of relationships to model complex relationships between multi-dimensional arrays based on data registration via the concept of advanced *index map relationships* (Sec. 3.2.4). The BRAINformat library and format also provides advanced support for definition, storage, and management of complex collections of data annotations (Sec. 3.3). We demonstrate the application of our framework to design a novel data standard for neuroscience data and its application to the storage and management of electrocorticography data collected during speech production (Sec. 4).

2 BACKGROUND AND RELATED WORK

The scientific community utilizes a broad range of data formats, which typically focus on different levels of the data organization and storage problem. Basic formats explicitly specify how data is laid out and formatted in binary or text data files (e.g., CSV, BOF, etc). While such basic formats are common in practice, they generally suffer from a lack of portability, scalability and a rigorous specification. For text-based files, languages and formats, such as the Extensible Markup Language (XML) [2] or the JavaScript Object Notation (JSON) [1], have become popular means to standardize documents for interchange of data. XML, JSON and other text-based data standards (in combination with character-encoding schema, e.g, ASCII or Unicode) play a critical role in practice in the exchange of usually relatively small, structured documents but are impractical for storage and exchange of large scientific data arrays due to the large overheads and cost they entail for storage, transfer, and I/O.

For storage of large-scale scientific data, HDF5 [14] and NetCDF [10] among others, have gained wide popularity. HDF5 is a data model, library, and file format for storing and managing large and complex data. HDF5 defines a set of core data object primitives, specifically: i) *Groups* which are similar to folders on a file system, used to group data objects, ii) *Datasets* which define n-dimensional arrays of arbitrary shape and data type, and iii) *Attributes* which are small meta data objects describing the nature and/or intended usage of groups or datasets. These data object primitives in combination provide the foundation for the organization and storage of highly complex data. HDF5 is portable, scalable, self-describing, and extensible and is optimized for storage and I/O of large-scale data. HDF5 is widely supported across programming languages and systems—e.g. R, Matlab, Python, C, Fortran, VisIt, ParaView etc.—and the HDF5 technology suite includes tools and applications for managing, manipulating, viewing, and analyzing data in the HDF5 format. HDF5 has been adopted as a base format across a broad range of application sciences, ranging from physics to bio-sciences and beyond[1]. Self-describing formats like HDF5 address the critical need for standardized storage and exchange of complex and large scientific data.

Even when self-describing formats like HDF5 are used, the organization of data—such as the structure, names, and descriptions of storage objects, e.g., groups, datasets or attributes—often still differ between applications and experiments. This diversity makes the development of common and reusable tools for processing, exchange, analysis, and visualization of data challenging. Some formats, e.g., VizSchema [12] and XDMF [3], propose to bridge this gap between general-purpose, self-describing formats and the need for standardized tools for data exchange, processing, and interpretation by augmenting HDF5 via lightweight, low-level schema (often based on XML) to further describe the organization of data. For example, the primary goal of XDMF (eXtensible Data Model and Format) [16, 3] is to help standardize methods to exchange scientific data between high-performance computing codes and tools. XDMF distinguishes and separates so-called light and heavy data. Light data contains the basic description of data arrays—e.g, the value type (float, integer, etc.), precision, location, rank, and dimensions of data arrays—while heavy data refers to the actual multi-dimensional arrays storing scientific data values. XDMF stores light data in XML while heavy data is stored in HDF5. The focus of formats like XDMF and VizSchema is primarily the standardized description of the low-level data organization to facilitate data exchange and tool development.

In contrast to XDMF and VizSchema, application oriented formats generally focus on specifying the organization of data in a semantically meaningful fashion, including but not limited to: the specification of storage object names, locations, descriptions, and data hierarchies. Many scientific application formats build on existing self-describing formats (e.g, HDF5), and examples include the NeXus [8] format for neutron, x-ray, and muon data, the OpenMSI format for mass spectrometry imaging data [11], the CXIDB format [9] for coherent x-ray imaging and many others. Application formats are often described by documents that specify the precise location and names of data items and in many cases provide some form of application-programmer interface (API) to facilitate reading and writing of format files. Some formats are further governed by formal, computer-readable, and verifiable specifications. For example, NeXus uses NXDL[2], an XML-based format and schema that allows scientists to define

[1] HDF users – `https://www.hdfgroup.org/HDF5/users5.html`

[2] NeXus Definition Language (NXDL) – `http://download.nexusformat.org/doc/html/nxdl.html`

the nomenclature and arrangement of information in a NeXus data file. On the level of HDF5 groups, NeXus also uses the notion of *Classes* to define the fields that a group should contain in a reusable and extensible fashion.

The critical need for data standards in neuroscience research has been recognized by several efforts over the course of the last several years; however, much work remains. Here, our goal is to contribute to this discussion by instantiating a usable and sustainable data standard for neuroscience research. The developers of the *Klustakwik* suite[7, 6] have proposed an HDF5-based data format for storage of spike sorting data. *Orca* (also called *BORG*) is an HDF5-based format developed by the Allen Institute for Brain Science designed to store electrophysiology and optophysiology data [3]. The *NIX* [13] project has developed a set of standardized methods and models for storing electrophysiology and other neuroscience data together with their metadata in one common file format based on HDF5. Rather than an application-specific format, NIX defines highly generic models for data as well as for metadata that can be linked to terminologies (defined via *od*ML) to provide a domain-specific context for elements. The *open metadata Markup Language od*ML [5] is a metadata markup language based on XML with the goal to define and establish an open and flexible format to transport neuroscience metadata. NeuroML [4] is also an XML-based format with a particular focus on defining and exchanging descriptions of neuronal cell and network models. The neurodata without borders (NWB)[4] initiative is a recent project with the goal *"[...] to produce a unified data format for cellular-based neurophysiology data based on representative use cases initially from four laboratories – the Buzsaki group at NYU, the Svoboda group at Janelia Farm, the Meister group at Caltech, and the Allen Institute for Brain Science in Seattle."* Members of the NIX, KWIK, Orca, BRAINformat, and other development teams[5] have been invited and have contributed to the NWB effort. NWB has adopted concepts and methods from a range of these formats, including from the here-described BRAINformat.

3 STANDARDIZING SCIENTIFIC DATA

3.1 Data Organization and File Format API

BRAINformat adopts HDF5 as its main storage backend and uses the following primary storage primitives to organize data within files:

- **Group:** A group is used—similar to a folder or directory on a file system—to group zero or more storage objects.
- **Dataset:** A dataset defines a multidimensional array of data elements, together with supporting metadata (e.g., shape and data type of the array).
- **Attribute:** Attributes are small datasets that are attached to primary data objects (i.e., groups or datasets) and are used in practice to store additional metadata to further describe the corresponding data object.
- **Dimension Scale:** This is a derived storage primitive that uses a combination of datasets and attributes to associate datasets with the dimension of another dataset. Dimension scales are used in practice to further characterize the dimensions of a dataset by describing, for example, the time when samples were measured or the location of samples in space.
- **Relationship Attributes:** Relationship attributes are a novel, custom attribute-type storage primitive that allows us to describe and model structural and semantic relationships between primary data objects in a human-readable and computer-interpretable fashion (described later in Section 3.2).

Neuroscience research inherently relies on complex collections of data from many modalities and sources. Examples include neural recordings, audio and video recordings, eye-tracking, motion tracking, task contingencies, external stimuli, derived analytic results, and many others. It is therefore critically important to specify formats in a modular and extensible fashion while enabling users to easily reuse format modules and integrate new ones. The concept of managed objects, which we will describe next, allows us to address this central challenge in an easy-to-use and scalable fashion.

[3] Orca slides presented at NWB: `http://crcns.org/files/data/nwb/h1/NWBh1_09_Keith_Godfrey.pdf`

[4] Neurodata without Borders – `https://crcns.org/NWB`

[5] `http://crcns.org/NWB/hackathon-1`

3.1.1 Managed Objects

A managed object is a primary storage object—i.e., file, group, or dataset—with: **1)** a formal, self-contained format specification that describes the storage object and its contents (see Section 3.1.2), **2)** a specific managed type/class, **3)** a human-readable description, and **4)** an optional unique object identifier, e.g., a DOI. In file, these basic managed object descriptors are stored via standardized attributes. Managed object types may be composed—i.e., a file or group may contain other managed objects—and further specialized through the concept of inheritance, enabling the independent specification and reuse of data format components. The concept of managed objects significantly simplifies the file format specification process by allowing larger formats to be specified in an easy-to-manage iterative manner. By encapsulating semantic sub-components, managed objects provide an ideal foundation for interacting with data in a manner that is semantically meaningful to applications.

The BRAINformat library provides dedicated base classes to assist with the specification and development of interfaces for new managed object types. The $ManagedObject$ base API implements common features to **1)** define the specification of a given managed type, **2)** recursively construct the complete format specification, automatically resolving nesting of managed objects, **3)** verify format compliance of a given HDF5 object, **4)** provide access to all common managed object descriptors stored in file (i.e., type, description, specification, and object identifier), and provides a standardized interface to **5)** access contained objects (e.g, datasets, groups, managed object etc.) from file, **6)** retrieve all managed object instances of a given managed type, and **7)** create appropriate manager class instances for a given HDF5 object based on the objects managed type.

In addition, the $ManagedObject$ base API defines and implements a standardized approach for creation of specific instances of managed objects stored in file via a common $create(..)$ method. Managed groups and datasets may be stored either directly within the parent managed group or created externally in a separate $ManagedObjectFile$ file storage container and included in the parent via an external link. In this way, the API directly supports self-contained and modular data storage in a transparent fashion. Self-contained storage eases data sharing, as all data is contained within a single file, while modular storage allows us to more easily manage file sizes and reduce the risk for file corruption by minimizing changes to existing files. From a user's perspective, modular and self-contained storage are handled transparently, i.e., a user can interact with managed objects in the same manner independent of whether the object is stored internal or external to the current HDF5 file.

To implement a new managed object type, a developer simply needs to define a new class that inherits from the appropriate base managed class type—i.e., $ManagedFile$, $ManagedGroup$, and $ManagedDataset$—and implement: **1)** the class method $get\_format\_specification(...)$ to create a formal format specification document (described next in Sec. 3.1.2) and **2)** the object method $populate(...)$, which is called by the standardized $ManagedObject.create(...)$ method and is used to implement the type-specific population of managed storage objects to ensure format compliance upon creation—i.e., the goal is to avoid that managed objects can be created in an invalid, non-format-compliant state to ensure that files remain format compliant throughout their life cycle.

3.1.2 Format Specification

To enable the broad application and use of data formats, it is critical that the underlying data standard is easy to interpret by application scientists as well as unambiguously specified for programmatic interpretation and implementation by developers. Therefore, each data format component (i.e, managed object type) is described by a formal, self-contained format specification that is computer interpretable while at the same time including human-readable descriptions of all components.

We generally assume that format specifications are minimal, i.e., all file objects that are defined in the specification must adhere to the specification, but a user may add user-defined data objects (i.e., groups, datasets, attributes etc.) to a file without violating format compliance. The relaxed assumption of a minimal specification ensures on the one hand that we can share and interact with all format-compliant files and file components in a standardized fashion, while at the same time enabling users to easily integrate dynamic and custom data (e.g, instrument-specific metadata), allowing researchers to save all their data using BRAINformat even if the current file standard should only partially cover the specific use-case. This is critical to enable

scientists to easily adopt the file standard and to allow the file standard to adapt to the ever-evolving experiments, methods, and use-case in neuroscience and facilitate new science rather than impeding it.

The BRAINformat library defines format specification document standards for the specification of the format of **1)** files, **2)** groups, **3)** datasets, **4)** attributes, **5)** dimension scales, **6)** managed objects, and **7)** relationship attributes. All specification documents are based on hierarchically composed Python dictionaries that can be serialized as JSON documents for persistent storage and sharing. For all data objects we specify the name and/or prefix of the object, whether the object is optional or required, and provide a human-readable textual description of the purpose and content of the object. Depending on the object type (e.g, file, group, dataset, attribute, etc.) additional information is specified, e.g., i) the datasets, groups, and managed objects contained in a group or file, ii) attributes for datasets, groups and files, iii) dimension scales of a dataset, iv) whether a dataset is a primary dataset for visualization and analysis or iv) relationships between objects among others. Figure 1 shows as an example an abbreviated summary of the format specification of our proposed data standard for neuroscience (described later in Section 4). Supplement 2 provides a more detailed discussion and examples of our format specification model. Relationship attributes and their specification are discussed in detail later in Sec. 3.2.

The BRAINformat library implements a series of dedicated data structures to assist with the development and interaction with format specifications. Using the provided data structures helps ensure that the generated documents are valid—e.g., that all required keys are set and that only valid keys and values are included in a document—and supports the incremental creation of format specifications, allowing the developer to step-by-step define and compose format specifications—similar to how one typically creates HDF5 files. For example, the following simple code can be used to generate the parts of the *BrainDataECoG* specification shown in Fig. 1:

```
>>> from brain.dataformat.spec import *
>>> # Define the raw dataset and associated attribute and dimension
>>> raw_data_spec = DatasetSpec(dataset='raw_data', prefix=None, optional=False, primary='True',
                                description="Dataset with the ECoG recordings data")
>>> raw_data_spec.add_attribute( AttributeSpec(attribute='unit', prefix=None, value='Volt') )
>>> raw_data_spec.add_dimension( DimensionSpec(name='space', unit='id', dataset='electrode_id',
                                           axis=0, description="Id of the recording electrode"))
>>> # Define the group and add the dataset
>>> brain_data_ecog = GroupSpec(group=None, prefix='ecog_data_',
                                description="Managed group for storage of raw ECoG recordings.")
>>> brain_data_ecog.add_dataset(raw_data_spec, 'ecog_data')
```

Using the BRAINformat specification infrastructure we can easily compile a complete data format specification document that lists all managed object types and their format. For example, the simple Python code shown here compiles the format specification document for our neuroscience data format directly from the Python API of our format (see also Sec. 4):

```
>>> from brain.dataformat.spec import FormatDocument
>>> import brain.dataformat.brainformat as brainformat
>>> json_spec = FormatDocument.from_api(module_object=brainformat).to_json()
```

Figure 1 shows an abbreviated summary of the result of the above code. The full JSON document is shown in Supplement 2, pp. 51 – 61. Alternatively, we can also recursively construct the complete specification for a given managed object type —e.g., here for the main file of the proposed neuroscience format described in Sec. 4— via:

```
>>> from brain.dataformat.brainformat import BrainDataFile
>>> from brain.dataformat.spec import *
>>> format_spec = BrainDataFile.get_format_specification_recursive()  # Construct the document
>>> file_spec = BaseSpec.from_dict(format_spec)                       # Verification of the document
>>> json_spec = file_spec.to_json(pretty=True)                        # Convert the document to JSON
```

```
{
    "BrainDataFile": {...},
    "BrainDataMultiFile": {...},
    "BrainDataData": {
        "attributes": [],
        "datasets": {},
        "description": "Managed group for storage of brain data (internal and external).",
        "group": "data",
        "groups": {},
        "managed_objects": [{"format_type": "BrainDataInternalData", "optional": false},
                            {"format_type": "BrainDataExternalData", "optional": false}],
        "optional": false,
        "prefix": null,
        "relationships": []
    },
    "BrainDataInternalData": {...},
    "BrainDataECoG": {
        "attributes": [],
        "datasets": {
            "ecog_data": {
                "attributes": [{"attribute": "unit", "optional": false,
                                "prefix": null,"value": "Volt"}],
                "dataset": "raw_data",
                "description": "Dataset with the ECoG recordings data",
                "dimensions": [{"axis": 0,
                                "dataset": "electrode_id",
                                "description": "Id of the recording electrode",
                                "name": "space",
                                "optional": false,
                                "relationships": [],
                                "unit": "id"},
                               ...
                ],
                "dimensions_fixed": true,
                "optional": false,
                "prefix": null,
                "primary": true,
                "relationships": []
            },
            ...
        },
        "description": "Managed group for storage of raw ECoG recordings.",
        "group": null,
        "groups": {},
        "managed_objects": [...],
        "optional": false,
        "prefix": "ecog_data_",
        "relationships": []
    },
    "BrainDataECoGProcessed": {...},
    "AnnotationDataGroup": {...},
    "BrainDataExternalData": {...},
    "BrainDataDescriptors": {...},
    "BrainDataDynamicDescriptors": {...},
    "BrainDataStaticDescriptors": {...},
    "ManagedObjectFile": {...},
}
```

Legend

`...`	This part has been omitted from the document. See Supplement 2 pp. 51–61 for details.
`BrainData`	Managed Object Type
`{...}`	Format specification

Figure 1. Abbreviated specification document for our neuroscience data format listing all current managed object types and partial specification for select structures illustrating the general structure of a formal specification document generated using the BRAINformat library. The full specification document is available as part of Supplement 2 pp. 51 – 61 (and the full recursive specification for a brain format file is shown in Supplement 2 pp. 35 – 51).

In this case, all references to other managed objects are automatically resolved and their specification is directly embedded in the resulting specification document. While the basic specification for $BrainDataFile$ consists only of ≈14 lines of code (see Supplement 2, pp. 30), the full, recursive specification contains more than 910 lines (see Supplement 2, pp. 35 – 51), illustrating the critical importance for being able to incrementally define format specifications.

The ability to compile complete format specification documents directly from data format APIs allows developers to easily integrate new format components (i.e. managed object types) in a self-contained fashion simply by adding a new API class

without having to maintain separate format specification documents. Furthermore, this strategy avoids inconsistencies between data format APIs and specification documents since format documents are updated automatically.

The concept of managed objects in combination with the format specification language and API provide an application-independent design concept that allows us to define application-specific formats and modules that are build on best practices.

3.2 Modeling Data Relationships

Neuroscience data analytics often rely on complex structural and semantic relationships between datasets. For example a scientist may use audio recordings to identify particular speech events during the course of an experiment and in turn needs to locate the corresponding data in an electrocorticography recording dataset to study the neural response to the speech events. In addition, we often encounter structural relationships in data, for example, in the case of data structures where one array indexes another array or two arrays share data dimensions because they have been acquired using the same recording device and many others. To enable efficient analysis, reuse, and sharing of neuroscience data it is critical that we can model the complex relationships between data objects in a structured fashion to enable human and computer interpretation and use of data relationships.

Modeling data relationships is not well-supported by traditional data formats, but is typically closer to the domain of scientific databases. In HDF5, we can compose data via HDF5 links (soft and hard) and associate datasets with the dimensions of another dataset via the concept of dimension scales. However, these concepts are limited to very specific types of data links that do not describe the semantics of the relationship. A new general approach is needed to describe more complex semantic links between data objects in HDF5.

3.2.1 Specifying and Storing Relationships

Here we introduce the novel concept of *relationship attributes* to describe complex semantic relationships between a source object and a target data object in a general and extensible fashion. Relationship attributes are associated with the source object and describe how the source is related to the target data object. The source and target of a relationship may be either a HDF5 group or dataset.

Relationship attributes are—like other file components— specified via a JSON dictionary and are part of the specification of datasets and groups. Like any other data object, relationships may also be created dynamically to describe any relationships that are unknown *a priori*. Specific instances of relationships are stored as attributes on the source HDF5 object, where the value of the attribute is the JSON document describing the relationship. As illustrated in Fig. 2, the JSON specification of a relationship consists of the following main components:

1. The specification of the name of the attribute and whether the attribute is optional. When stored in HDF5 we prepend the prefix $RELATIONSHIP\_ATTR\_$ to the user-defined name of the attribute to describe the attribute's class and ease identification of relationship attributes.
2. A human-readable description of the relationship and an optional JSON dictionary with additional user-defined data relevant to the relationship.
3. The specification of the type of the relationship (described next in Sec. 3.2.2).
4. The specification of the axes of the source object to which the relationship applies. This may be: i) a single index, ii) a list of axes, iii) a dictionary of axis indices if the axes have a specific user meaning, or iv) None if the relationship applies to the source object as a whole. Note, we do not need to specify the location of the source object, as the specification of the relationship is always associated with either the source object in HDF5 itself or in the format specification.
5. The specification of the target object describing the location of the object and the axes relevant to the relationship (using the same relative ordering or names of axes as for the source object).

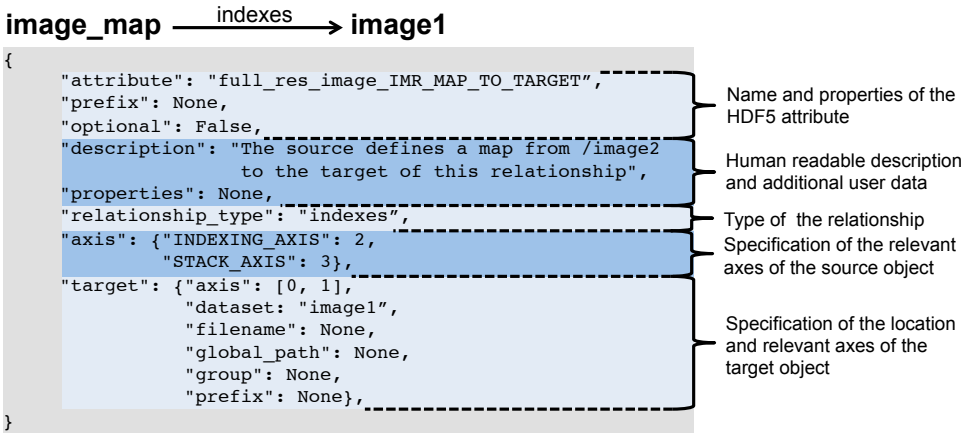
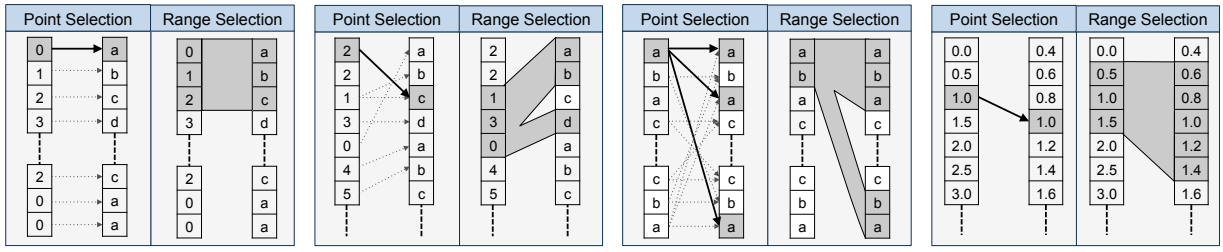


Figure 2. Example specification of a relationship attribute illustrating the main components of the specification.

3.2.2 Relationship Types

The relationship type describes the semantic nature of the relationship. The BRAINformat library currently supports the following main types of relationships, and additional types can be added in the future:

- **order:** This relationship type indicates that elements along the specified axes of the relationship are ordered in the target in the same way as in the source. This type of relationship is very common in practice. For example, in the case of dimension scales, an implicit assumption is that the ordering of elements along the first axis of the scale-dataset matches the ordering of the elements of the dimension it describes. This assumption, however, is only implicit and is by no means always true (nor does HDF5 require this relationship to be true). Using an *order* relationship we can make this relationship explicit. Other common uses of *order* relationships include describing the matched ordering of electrodes in datasets that have been recorded using the same device or matched ordering of records in datasets that have been acquired synchronously.
- **equivalent:** This relationship type expresses that the source and target object encode the same data (even if they might store different values). This relationship also implies that the source and target contain the same number of values ordered in the same fashion. This relationship occurs in practice any time the same data is stored multiple times with different encodings. For example to facilitate data processing a user may store a dataset of strings with the names of tokens and store another dataset with the equivalent integer ID of the tokens.
- **indexes:** An indexes relationship describes that the source dataset contains indices into the target data object (group or dataset). In practice this relationship type is used to describe basic data structure where we store, for example, a list of unique values (tokens) along with other arrays that reference that list.
- **shared encoding:** This relationship indicates that the source and target data object contain values with the same encoding so that the values can be directly compared (via equals "=="). This relationship is useful in practice any time two data objects (datasets or groups) contain data with the same encoding (e.g. two datasets describing external stimuli using the same ontology).
- **shared ascending encoding:** This relationship type implies that the source and target data object share the same encoding and in addition that the values are sorted in ascending order in both data objects. The additional constraint on the ordering enables i) comparison of values via greater than ">" and less than "<" (in addition to equals ==) and ii) more efficient processing and comparison of data ranges. For example, in the case of two datasets that encode *time*, we often find that individual time points do not match exactly between the source and target (e.g, due to different sampling rates). However, due to the ascending ordering of values, a user is still able to compare ranges in *time* in a meaningful way.



(a) order (b) indexes (c) shared encoding / indexes values (d) shared ascending encoding

Figure 3. Overview of the main relationship types and the implied mapping of point- and range-based selections from the source to the target object. In each cell we show the source object on the left and the target object of the relationship on the right. **(a)** For *order* relationships we can directly map array indices between the data objects. In the case of *order* relationships involving HDF5 Groups we assume alphabetic ordering of elements. **(b)** In the case of *indexes* relationships we map selections by retrieving the relevant indicies from the source array. **(c)** For *shared encoding* and *indexes values* relationships we support data selection via value-based data mapping, i.e., we map selections by locating all data values in the target object that match at least one of the values we selected in the source object. **(d)** *Shared ascending encoding* relationships behave in general similar to *shared encoding* relationships, however, the additional constraint that values are sorted in ascending order enables us to map range selections directly based on the minimum and maximum value selected in the source dataset (in contrast to the strict equal value matching of *shared encoding*). **(e)** *User* relationships define custom user semantics and do not imply a specific mapping between data elements (not shown).

- **indexes values:** This relationship is typically used to describe value-based referencing of data and indicates that the source data object selects certain parts of the target data object based on data values (or keys in the case of groups). This relationship is a special type of *shared encoding* relationship.
- **user:** The *user* relationship is a general container to allow users to specify custom semantic relationships that do not match any of the existing relationship patterns. To further characterize the relationship, we often store additional metadata about the relationship as part of the user-defined *properties* dictionary of the relationship attribute.

3.2.3 Using Relationship Attributes

Relationship attributes are a direct extension to the previously described format specification infrastructure. Similar to other main data objects, BRAINformat provides dict-like data structures to help with the formal specification of relationship attributes. In addition, the BRAINformat library also provides a dedicated *RelationshipAttribute* API, which supports creation and retrieval of relationship attributes (as well as index map relationship, described in Sec. 3.2.4) and provides easy access to the source and target HDF5 object and corresponding specifications of relationship attributes.

One central advantage of explicitly defining relationships is that it allows formalizing the interactions and collaborative usage of related datasets. In particular, the relationship types imply formal rules for how to map data selections from the source object of a relationship to the target object. The *RelationshipAttribute* API implements these rules and supports slicing, which allows us to easily map selections from the source to the target data object using the same familiar slicing syntax. For example, assume we have two datasets A and B related to each other via an $indexes$ relationship $R_{A \to B}$. A user now selects the values $A[1:10]$ in the source dataset A and wants to locate the corresponding data values in the target data object B. Using the BRAINformat API we can now simply write $R_{A \to B}[1:10]$ to map the selection $[1:10]$ from the source A to the target B, and if desired retrieve the corresponding data values in B via $B[R_{A \to B}[1:10]]$. Figure 3 provides an overview of the rules for mapping selections based on the type of the relationship.

Relationship attributes standardize the specification, storage, and programmatic interface for creating, discovering, and using relationships and related data objects. Describing relationships between data explicitly greatly simplifies the process of interacting with multiple datasets and facilitates the collaborative use of data by enabling utilization of multiple datasets in conjunction without having to *a priori* know the relationships and datasets involved. In this way, relationship attributes also open the route for the standardized development of novel data-driven analytics and workflows based on the programmatic discovery and use of related data objects.

3.2.4 Index Map Relationships

Beyond the description of direct object-to-object relationships, relationship attributes also form the building blocks that allow us to specify higher-order relationships. Using relationship attributes we can define chains of object-to-object relationships that, when interpreted in conjunction, express highly complex structural and semantic relationships. For example, imagine the following situation. Scientists have acquired an optical microscopy image A and an electron microscopy image B of the same brain. Using the optical image a scientist identifies a particular brain region of interest and now wants to study the same region further using the electron-microscopy image. This seemingly simple task of accessing corresponding data values in two related datasets is in practice, however, often highly complex. Even if the data registration problem between the datasets is solved, a user still has to know exactly: i) the location of both datasets A and B, ii) how the two datasets are related, iii) what the transformations generated by the data registration process are, iv) how to utilize that information to map between A and B, and v) write complex, custom code to access the data.

Index map relationships allow us to explicitly describe this complex relationship between A and B via a simple chain of object-to-object relationship attributes and to greatly simplify the cooperative interaction with the data. Rather than describing the relationship between A and B directly, users can create an intermediate index map $M_{A\to B}$ that stores for each pixel in A the index of the corresponding pixel(s) in B. $M_{A\to B}$ explicitly and unambiguously describes the mapping from A to B so that we can directly utilize the mapping without having to perform complex and error-prone index transformations (which would be needed if we described the mapping implicitly, e.g., via scaling, rotation, morphing and other data transformations). As the table in Fig. 4 shows, via a simple series of relationship attributes describing simple object-to-object relationships, we can unambiguously describe the complex relationship between A and B via $M_{A\to B}$. Given only our source dataset A (or index map $M_{A\to B}$) we can now easily discover all relevant data objects (A, B, and $M_{A\to B}$) and relationships (Fig. 4) without having to *a priori* know the mapping or the location of the datasets. Via the index map relationship, we can now directly map selections: i) from A to $M_{A\to B}$ and *vice versa* ii) from $M_{A\to B}$ to B, and most importantly iii) from A to B simply by slicing into our $indexes$ relationship (Fig. 4, row 3) to retrieve the corresponding indices from our index map $M_{A\to B}$. As data mappings are described explicitly, index map relationships enable registration and mapping under arbitrary transformations. Also, mappings are not required to be unique—i.e., arbitrary N-to-M mappings between elements are permitted—and the source and target of relationships may not just be datasets but also groups, i.e., index map relationship can be used to define mappings between contents of groups or even groups and datasets in HDF5.

	Source	Relationship	Target	Description
1.	A	$\xrightarrow{order}$	$M_{A\to B}$	This relationship describes that elements in A are ordered in the same way as the elements in the index map $M_{A\to B}$. In addition we may further specify the axes in the source A and target $M_{A\to B}$ along which the relationship applies.
2.	A	$\xleftarrow{order}$	$M_{A\to B}$	Inverse of (1), describing the object ordering relationship between $M_{A\to B}$ and A.
3.	$M_{A\to B}$	$\xrightarrow{indexes}$	B	This relationship indicates that $M_{A\to B}$ stores indices into B and describes how our map can be used to access B. An example specification of this relationship is shown in Fig. 2.
4.	A	$\xrightarrow{user}$	B	An optional user-type relationship may be used to further characterize the semantic relationship between A and B.

Figure 4. Overview of the relationships used to define an advanced *index map relationship*. We present a specific example later in Fig. 5.

BRAINformat implements the concept of index map relationships—similar to dimension scales and relationship attributes—via a set of simple naming conventions for the attribute names. In addition to the *RELATIONSHIP_ATTR* prefix, we use a set of reserved post-fix values—specifically *_IMR_MAP_TO_TARGET*, *_IMR_MAP_TO_SOURCE*, *_IMR_SOURCE_TO_MAP*, *_IMR_SOURCE_TO_TARGET*—that are appended to the user-defined attribute name to identify the different components of the index map relationship. The BRAINformat API directly supports index map relationships so that we can, for example, directly

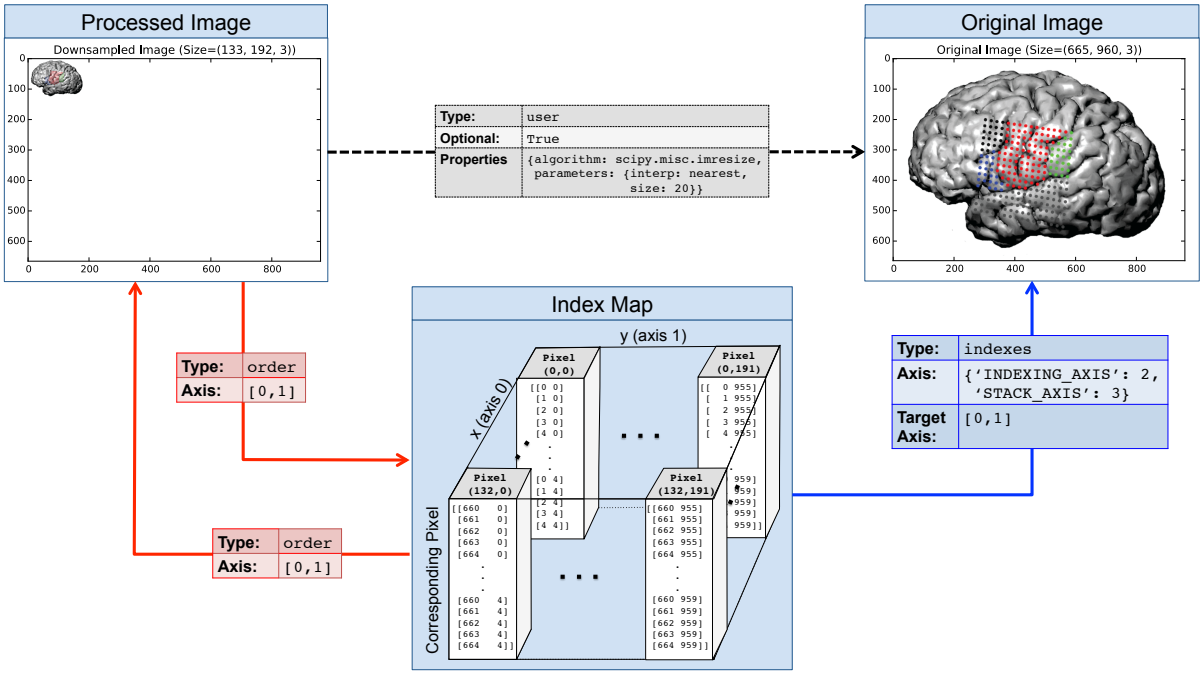
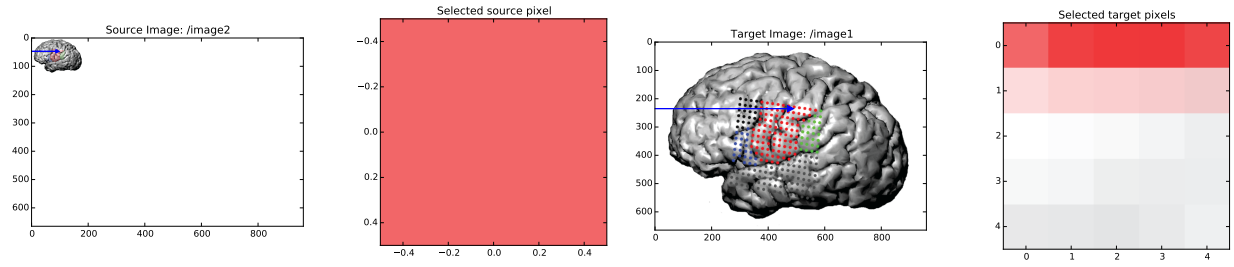


Figure 5. Illustration of an index map relationship describing the interaction between a processed image and the original image. The processed image is in this case a 5× smaller version of the original image created using nearest neighbor interpolation. The intermediary index map describes for each pixel in the processed image which pixels it corresponds to in the the original image. Two *order* relationships (red arrows) describe the interactions between the processed image and the map and vice versa. A third *indexes* relationship links our index map to the original image and describes how the map can be used to access the original image. Optionally, we may create a fourth *user* relationship (black arrow) to further characterize the semantic relationship between the processed and original image (e.g, to store a description of the algorithm and parameters used to generate the image). Naturally, we can also describe the inverse mapping between the original and processed image via a second index map relationship.

create and locate all relationships that define an index map relationship via a single function call and programmatically interact with the relationships. Supplement 1 (pp.12–26) includes an overview and basic tutorial of the API for creating and using index map relationships.

Index map relationships have broad practical applications, including data registration, sup-component analysis, correlation of data dimensions, and optimization. Index map relationships are directly applicable to specify the mapping between images in a time series or a stack of physical slices as well as to define correspondences between images from different modalities. We may also define mappings between select dimensions of a dataset to correlate data from different recordings in time or space. Furthermore, analytics are often based on characteristic sub-components of a dataset. As such, a user may extract and separately process sub-components of datasets (e.g. a sub-image of a single cell) and use index map relationships to map the extracted or derived analysis data back to the original data. To optimize data classification, feature detection, and other compute-intensive analyses, a user may perform initial calculations on lower-resolution versions of a dataset and use index map relationships to access corresponding data values in the high-resolution version of the dataset for further processing.

Figure 5 illustrates an example index map relationship for the latter use-case. The complete source code and further details for this example are available in Supplement 3. In this example, our original dataset is a RGB image dataset of size (665×960) that has been processed to reduce the size in the two spatial dimensions by a factor of 5 to (133×3) via nearest neighbor interpolation. Each pixel in the processed image, hence, maps to a 5×5 sub-region in the original image. We, hence, create a 4-dimensional index map dataset where: 1,2) the first two dimensions correspond to the spatial dimensions x and y of the images, 3) the third dimension is our index axis of length 2 since each pixel is described by two integer indices, and 4) the fourth dimension is our stacking axis with the list of all corresponding pixel. Using the BRAINformat API, we can now create the index map relationship—which is defined by the arrows shown in Figure 5—via a single function call (Supplement 3–Sec. 1.3).



(a) Processed Image (b) Original Image

Figure 6. Illustration of the result from using our index map relationship to perform data selection in our source, processed image (a) and target, original image (b). **(a)** First we apply the selection (47, 98) (blue arrow) to our source dataset (left). As expected, this results in the selection of a single pixel (right). **(b)** We next map the same selection to our target dataset (left). From the blue arrow we can see that the selection was mapped correctly to same relative location as in our source, image. The pixel plot (right) illustrates that the mapping resulted, as expected, in the selection of a 5×5 sub-image from our target image. We can also see that the top-left pixel of our selected sub-image matches the color of the pixel we retrieved in the source dataset (a, right). This is expected since the source image (a, left) was generated from the target image (b, left) via $5\times$ downsampling using nearest neighbor interpolation.

As illustrated in Figure 6, we can now easily map a selection (here $[47, 98]$) from our source (processed image) to the target (original) image simply by slicing into our index map relationship (imr) via $imr[MAP\_TO\_TARGET][47, 98]$ and retrieve the data of the corresponding subimage from our original image (Supplement 3–Sec. 1.4).

As this simple example illustrates, index map relationships allow us to explicitly describe complex relationships between data. Being able to unambiguously describe complex relationships is critical to enable us to programmaticaly utilize relationships and perform complex multi-data analytics and to reduce risk for errors due to implicit assumptions about relationships between data objects. Index map relationships are not restricted to just define relationships between HDF5 datasets but can also be used to define relationships involving HDF5 groups or managed objects. Here we focus on index map relationships, but the same basic concept of chaining relationships could be applied to construct other types of complex object inter-relationships as well.

3.3 Annotating Scientific Data

Advanced neuroscience analytics rely on complex data access patterns driven by data semantics. For example, common neuroscience data analytics often focus on understanding how different brain regions—measured, e.g., by collocated electrodes—operate together and interact with each other during specific, randomly interleaved events, e.g., time intervals when a subject said *'baa'* or performed a particular motion. The ability to annotate data by associating semantic metadata with data subsets is critical to facilitate these kinds of analyses. Annotating data in a scalable and usable fashion is challenging and relies on complex data structures to describe data selections and associated metadata.

To support data annotation, the BRAINformat library provides a series of modules that implement general and reusable data structures to describe individual data selections and data annotations (Sec. 3.3.1) and modules to manage and store collections of data annotations (Sec. 3.3.2). The BRAINformat annotation package supports annotation of in-memory data arrays (e.g., numpy arrays) as well as in-file arrays (i.e., HDF5 datasets) and processing of data annotations may be performed in-memory or out-of-core (i.e., with the majority of data residing on disk and being only loaded when needed).

3.3.1 Data Selection and Annotation

The first steps in annotating data is to describe 1) the data object that contains the data and 2) the data selection describing the subset of the data to annotate. The first part of describing the data object itself is generally simple and consists of either a basic reference to the data object in memory or an HDF5 link to the corresponding object in file. Describing data selections, however, is in practice not as simple. In the context of neuroscience data, researchers often need to generate a large numbers of annotations that refer to complex subsets of data, leading to advanced data selection, storage, and API requirements for describing, storing, and interacting with data selections. For example, along a single axis (such as time), features of interest

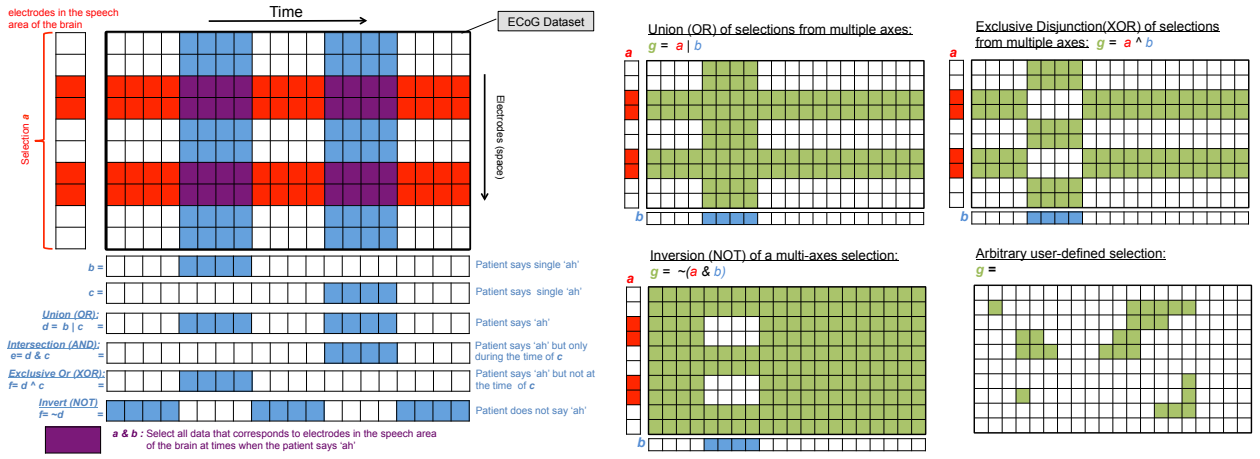
		Slice [start:stop:step]	Binary vector	List of indices	Word-aligned compressed bit vectors
Data selection requirements	Supports structured 1D selections	Yes	Yes	Yes	Yes
	Supports arbitrary 1D selections	No	Yes	Yes	Yes
	Supports common binary operations (AND, OR, NOT, XOR)	No	Yes	Yes	Yes
	Supports structured multi-dimensional selections	Yes	Yes	Yes	Yes
	Supports arbitrary multi-dimensional selections	No	Yes (when using full Boolean maps)	Yes (when using multi-dimensional indices)	Yes (extension to multi-dimensional compressed bitmaps possible but non-trivial)
Storage and API requirements	Low storage requirements	Yes	Highly compressible in file. Constant selection array size (n bytes per axis). Representing arbitrary, multi-dimensional selections depends directly on the size of the data object.	Grows linear with the number of selected elements and number of data dimensions. Compact for highly sparse selections. Expensive for dense selections.	Compact and highly compressed. Bitvectors can be merged and processed directly in compressed form.
	Multiple selections can be easily represented via a fixed number of arrays.	Yes	Yes	Yes, but additional array index schema are needed to represent varying size selection vector in a compact array data structure.	Yes, but additional array index scheme are needed to represent varying size selection vector in a compact array data structure.
	Selection vectors can be directly interpreted without the BRAIN format API	Yes	Yes	Yes	No
Summary	Main Advantages	Simple and easy-to-use	Fullfills most requirements	Fullfills most requirements	Highly efficient
		Constant selection vector size	Easy to use	Easy to use	
		Compact storage requirements	Constant selection vector size	Efficient for sparse selections	
		Easy to represent and use in file and memory	Compact storage due to high compressibility		
			Easy to represent and use in file and memory		
	Main Disadvantages	Suited for highly structured selections only	Expensive when having to keep many selections in uncompressed form in memory.	Expensive for dense selections.	Hard to use without dedicated API
			Expensive for arbitrary selections	Variable length selection vectors depending on selection size	Variable length selection vectors depending on selection size
					Extension to arbitrary selections can be non-trivial

Figure 7. High-level comparison of four common schema for representing data selections. In each case, structured, multi-dimensional selections are constructed by the intersection (AND) of one-dimensional selections along the individual axes while *None* is used to efficiently describe the selection of all elements along a given axis.

are often discontinues—e.g., when describing multiple events of the same type—and complex features are often the result of combinations of basic features along multiple dimensions—e.g., the output measured by electrodes located in the hippocampus while the animal is in a specific location.

The table in Fig. 7 provides a high-level comparison of four common schema for describing data selections (columns) with respect to their general behavior in regard to some main requirements for annotating neuroscience data (rows). Slicing is a very convenient way to express highly structured selections that can be described via a simple tuple of *(start, stop, step)* but it does not support selection of complex data subsets. Binary vectors—describing for each element along a given axis whether the element is selected—are generally a good option. One main disadvantage of binary vectors is that the memory cost can be high when having to process a large number of selections in uncompressed form in memory. In practice, however, most operations can be performed iteratively and out-of-core. Lists of indices—describing along each axis the specific, selected elements—are also a very good option. The main disadvantage of index lists lies in the high cost for describing dense selections and the variable length arrays needed to describe the selections. More advanced data selection methods, such as, word-aligned hybrid compressed bitmap indices [17, 18] are also very promising. One main disadvantage of such advanced indexing schema is that they are not easily interpreted without a dedicated API, potentially hindering reuse of the HDF5 files. For the initial development of the BRAINformat annotation API and format we have chosen binary vectors as the main scheme to represent complex data selections and are planing to add support for additional schema in the future.

Fig. 8(a) illustrates how we can represent and combine complex selections using binary vectors. Along each dimension we store a binary vector describing the elements that are selected (True, color) or not selected (False, white). This allows us to easily represent arbitrary selections using constant-length selection vectors. The binary vectors can be efficiently combined



(a) Basic data selections

(b) Complex data selections

Figure 8. Overview of common data selections and binary combinations of data selections. **(a)** Basic data selections are defined via 1D binary vectors demarking the elements selected along a given dimension. Basic multi-dimensional selections are then defined via binary AND (&) combinations of such per-axis binary vectors. This allows complex selections to be expressed and stored efficiently. Along a given axis we may combine selections via boolean operations—including, AND, OR, XOR, and NOT—without the need to expand the selection to a complex selection. **(b)** We support complex selections that cannot be expressed via combinations of hyper-slices through expansion of the selection to a full binary map allowing the definition of arbitrary selections. Complex selections are required in practice to define OR, XOR, and NOT combinations of multi-dimensional selections and for arbitrary user selections.

directly using common binary operations and multi-dimensional selections can be easily described by the intersection of multiple binary vectors. Arbitrary multi-dimensional selections—needed to describe complex, multi-dimensional combinations of basic selections and arbitrary user-defined selections—can be described via full binary selection masks (see Fig. 8(b)).

Now that we can describe data selections, we can extend our design to define annotations. A single data annotation in BRAINformat consists of the following main elements: **i)** the data selection object describing the data object and subregion of the data the annotation applies to, **ii)** a user-defined string indicating the type of the annotation, **iii)** a human-readable description of the annotation, and **iv)** a dictionary of additional user-defined properties of the annotation, Currently the format requires that the keys of the properties dictionary are strings and that the values are arbitrary, basic data objects, e.g, stings or numbers. This simple design allows us to describe complex annotations in an easy-to-use fashion.

The BRAINformat data selection and annotation API can be used to annotate any data object that can describe its shape as an n-dimensional array and supports numpy/h5py-style array slicing, including numpy arrays, HDF5 datasets, and certain managed objects that implement an array-like interface. The data selection and annotation API supports (among other things):

- Selection of elements via basic array slicing and assignment, e.g., to select the first five elements along the $time$ axis for a data selection A, we may write $A['time', 0:5] = True$.
- Retrieval of the selection vector along a given axis via simple slicing, e.g, $A['time']$.
- Retrieval of the data selected by an annotation or data selection via $A.data()$.
- Common binary operations to merge data selections and annotations, including i) AND, ii) OR, iii) NOT, and iv) XOR.
- Comparison of data selections and annotations via common operations, such as $>$, $>=$, $<$, $<=$, $==$, $!=$, and in. These operations are based on the comparison of the selected array indices so that, e.g., $A > B$ is only true if B is a true subset of A (in contrast to a simple length-based comparison which would only require $|A| > |B|$).
- Preceded ($A << B$) and follows ($A >> B$) operations describing whether all elements selected by A have array indices less than or greater than B, respectively. This is useful, for example, to identify if an event in $time$ selected by A occurs before/after B.

- Basic investigation via functions like len, $count$, $counts$, $axes$ or $axis\_bounds$ to retrieve the total and per-axis number of elements selected or the axes that are restricted and index ranges *et cetera*.

3.3.2 Managing Collections of Data Annotations

So far we have focused on describing single annotations. Neuroscience analytics often rely on large collections of annotations to describe, e.g., multiple behavioral measures, external stimuli, brain regions and many other types of annotations. During the course of an experiment we often encounter many thousands of events and features of a given type and in some cases millions (e.g, action potentials). Storing all these annotations individually would quickly result in an explosion of datasets and groups in the HDF5 file, hindering both usability and computational performance. It is, therefore, critical that we can represent collections of annotations in a compact fashion via a limited number of data objects (Fig. 7). In addition, we need to be able to easily search collections of annotations to locate subsets of annotations of interest, e.g., all annotations describing movements to a specific region in space.

Annotation collections are managed in the BRAINformat library by the $AnnotationDataGroup$ (and $AnnotationCollection$) module, which uses the $ManagedObject$ design (see Section 3.1.1) to define a general, reusable, and extensible storage module and format for collections of data annotations. Each collection of annotations applies to a specific data object and has a user-defined description. The corresponding binary selection vectors of all annotations in a given collection have the same length, since all annotations refer to the same data object. For each data axis, we can store an arbitrary number of selections in a single two-dimensional data array with a shape of $\#selections \times \#values$. To reduce storage cost, we enable $gzip$ compression —which is natively supported by HDF5—when saving the binary data selection arrays to file. Using compression drastically reduces the size of data selections in file and enables us to efficiently store large collections of data annotations. The type, descriptions, and individual properties of all annotations are then stored separately in one-dimensional arrays. This simple scheme allows us to store an arbitrary number of annotations in a fixed number of arrays while allowing us to easily retrieve specific annotations as well as independently access individual fields for searching (e.g., annotation type, description, and individual properties).

Collections of annotations may be created in-memory or in-file. When accessing collections of annotations that are stored in-file, the bulk of the data—such as the selection properties and binary selection vectors—typically remain out-of-core and are only read when needed,for example when searching for annotations based on a particular property. To ease the use of collections of annotations, the BRAINformat API supports:

- **Filtering** (i.e., search) of annotations to locate annotations based on the:
  i) index of annotations,
  ii) axes that are restricted by the annotations to find, e.g., all annotations that select features in $time$,
  iii) full or partial type of annotations to find, e.g., all annotations that define a *speech event*,
  iv) full or partial description of annotations, and
  v) full or partial user-defined metadata properties of annotations to find e.g., all annotations that have a particular user-level, start time or stop time etc..

  All filter functions return one-dimensional binary selection vectors that can be easily combined via standard binary operations. This allows us to easily define complex queries. For example to locate all *speech events* when a subject said *'baa'* in a collection of annotations C, we can simply write $C.type\_filter('speech\ event') \ \&\ C.property\_filter(key =' vocalization', value =' baa')$.
- **Selection** of subsets of annotations, i.e., the creation of new collections of annotations through the application of a filter via basic slicing, e.g., $C[\, C.type\_filter('speech\ event')]$.
- **Merging** of all annotations in a collection to a single annotation via union (OR), intersection (AND), and exclusive disjunction (XOR).

- **Loading/retrieval** of the individual annotations and data selected by the annotations in the collection.
- **Basic introspection** to retrieve information about, e.g., the number of annotations, list of unique annotation types and descriptions, properties, etc..
- **Creation, expansion, and saving** of annotation collections.

Other more specialized functions of annotation collections include the calculation of a containment matrix, describing which annotations are contained in each other. This is useful in the context of hierarchically organized annotations. For example, in the context of speech, we have annotations that describe individual $syllables$, $words$, $sentences$ and so on.

4 RESULTS

4.1 Applications to Neuroscience Data

In the following we describe the application of the BRAINformat data standardization framework to the development of a data format for neuroscience data with an initial focus on electrocorticography (ECoG) data collected from neurosurgical patients during speech production. This data shares many requirements with standard electrophysiology data collected by the broader neuroscience community: storage of voltage recordings over time across multiple spatial distributed sensors with heterogenous geometries, complex and multi-tiered task descriptions, post-hoc processing of raw data to extract the signal of interest, the association of physiology data with multi-modal data streams collected simultaneously by other devices, the linkage of data associated with the same 'task' across multiple sessions, and the necessity to store rich meta-data to make sense of it all.

4.1.1 High-level Data Organization

Fig. 9 shows and example visualization of a BRAINformat file using HDFView. The tree view shown on the left illustrates the high-level data hierarchy. In our discussion of the high-level data organization we use the following notation to denote the path in HDF5 and corresponding managed type: ***path* : *managed type***.

In the main HDF5 file ***/* : *BrainDataFile*** the data is organized in a basic semantic hierarchy. On the highest level we distinguish between data and descriptors, i.e., raw and processed data generated through experimentation and analysis vs. globally accessible metadata. We then further distinguish between static metadata (i.e. descriptions of the basic data acquisition and experimental parameters) and dynamic metadata (e.g. descriptions of post-processing parameters) and categorize data into internal data (i.e. data collected inside the brain, e.g. electrophysiology recordings) and external data (i.e. data collected external to the animal, e.g. sensory stimuli, audio recordings, position of body parts, etc.). These divisions are not strictly necessary, but impose some minimal structure on the format that eases the interpretability by users. The following list illustrates the high-level data organization in more detail:

- ***/data* : *BrainDataData*** contains the actual raw and processed data generated through experimentation and analysis.
    - ***/data/internal* : *BrainDataInternalData*** contains all internal data, i.e., all raw and processed data from physiological measurements.
        - ***/data/internal/ecog_data_#* : *BrainDataECoG*** is designed for storage of voltage recordings over time across multiple spatially distributed sensors with heterogenous geometries, and complex and multi-tiered task descriptions (see Sec. 4.1.2).
        - ***/data/internal/ecog_data_processed_#* :*BrainDataECoGProcessed*** is derived from ***BrainDataECoG*** and is designed for storage of post-processed voltage recordings over time across multiple spatial distributed sensors where signals of interest have been extracted and optionally categorized (see Sec. 4.1.3).

- */data/external* **:** ***BrainDataExternalData*** is used to collect all external data, such as recordings of sensory stimuli and other external measurements.

- */descriptors* **:** ***BrainDataDescriptors*** is a container for global metadata. Specific metadata objects may be referenced in other managed objects via HDF5 links. This strategy avoids redundant storage while at the same time providing easy access to the data from specific data groups and allowing scientists to collect general metadata in a central location, facilitating meta- and mega analysis.
  - */descriptors/static* **:** ***BrainDataStaticDescriptors*** is a container for static metadata, e.g., metadata describing the instruments and other fixed information.
  - */descriptors/dynamic* **:** ***BrainDataDynamicDescriptors*** is a container for dynamic metadata, e.g., information that is derived through post-hoc analyses or metadata that may dynamically change during the data life cycle.

In practice, scientists regularly acquire data in series of distinct experiment sessions often distributed over long periods of time. To facilitate management and sharing of data, it is useful to store the data generated from such distinct recordings in separate data files, yet for analysis purposes the data often needs to be analyzed in context. To allow the organization of related data files we support the grouping of files in container files */* **:*BrainDataMultiFile*** in which each primary ***BrainDataFile*** file is represented by an HDF5 group */entry_#* that defines an external link to the root group of the corresponding file. This simple concept enables users to interact with the data as if it were located in a single file while the data is physically being stored distributed across many files.

In addition to the format-specific modules described so far, we use the generic ***AnnotationDataGroup*** (see Sec. 3.3.2) managed type for management and storage of collections of annotations associated with raw data and processed data (Sec. 3.3.2). We also use the generic ***ManagedObjectFile*** module (see Sec. 3.1.1) to support modular storage of managed objects in separate HDF5 files (which are in turn included in the parent via an external links). This allows users to flexibly store and share analytics as independent files while at the same time making the results easily accessible from the main data file and limiting the need for large-scale updates to the main file.

We will next discuss the storage of voltage recordings over time across multiple spatial distributed sensors via the ***BrainDataECoG*** and ***BrainDataECoGProcessed*** modules in more detail. For further details on the data organization we refer the interested reader to the specification documents of the data format shown in Supplement 2 (pp 35 – 62). Figure 1 also shows an abbreviated version of the specification document, listing all current managed object types in blue.

4.1.2 Storing ECoG Data

A central application in neuroscience data is the acquisition and storage of voltage recordings over time across multiple spatial distributed sensors, e.g., via electrocorticography (ECoG), multi-channel electrophysiology from silicon shanks or Utah arrays. In the following we focus in particular on electrocorticography (ECoG) data collected from neurosurgical patients during speech production, however, we intend to extend these capabilities to other use-cases as well—such as physiology data collected in model species during standard sensory, motor, and cognitive neuroscience tasks—and the format has been designed with this extensibility in mind.

The ***BrainDataECoG*** module defines a managed group in HDF5 that serves as a container to collect all data pertaining to the voltage recordings in a single location. The primary dataset ***raw_data*** defines a two-dimensional, $space \times time$ array storing electrical recordings in units of $Volts$. Auxiliary information about the data, e.g, the ***sampling_rate***, in Hz, the ***unit*** of $Volts$, and the spatial ***layout*** of the electrodes are stored as additional datasets and attributes.

The ***raw_data*** is also further characterized via a series of dimensions scales describing: **1)** the identifier of electrodes (e.g. linear channel index from DAQ) (***electrode_id***), **2)** the sample time in milliseconds (***time_axis***), and optionally **3)** the anatomical name (***anatomy_name***) and integer id (***anatomy_id***) of the spatial region where each electrode is located. In addition, the

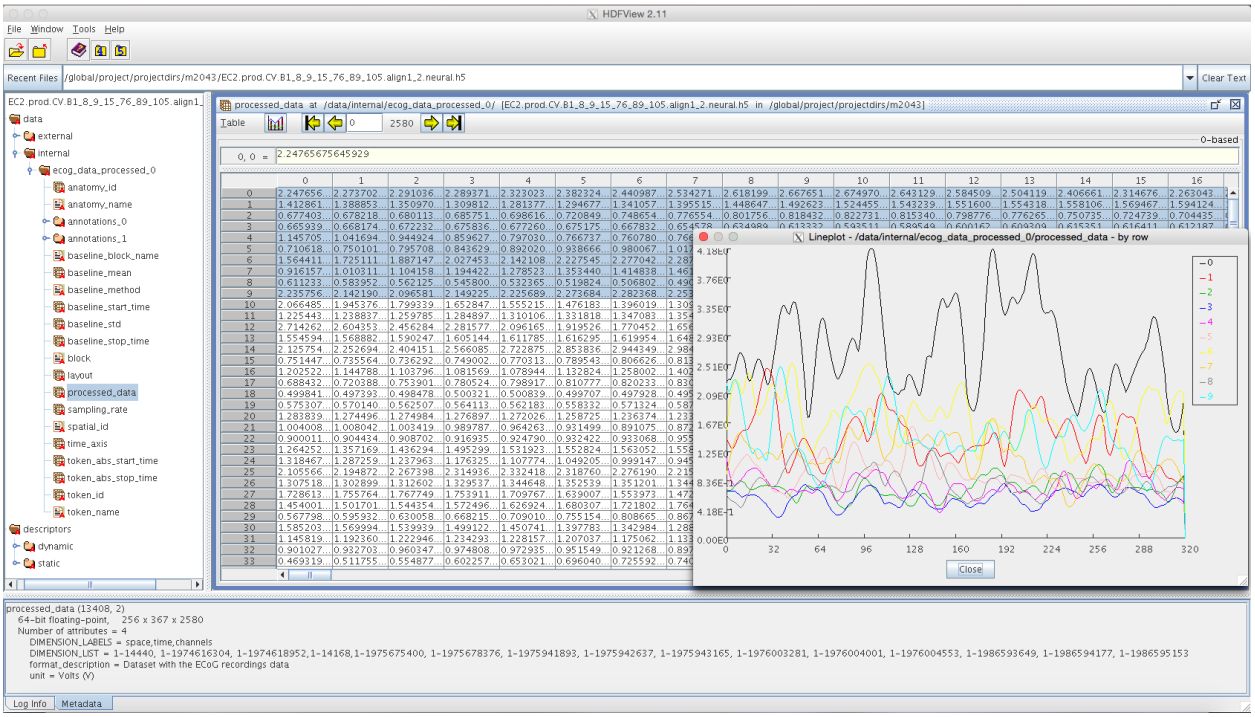


Figure 9. Example HDFView [15] visualization of a BRAINformat file. **Left:** Tree view of the basic data hierarchy. **Right:** Table view of a processed ECoG dataset and curve plot of the first 10 waveforms. **Bottom:** Summary of properties and attributes of the processed ECoG data array.

BrainDataECoG API provides convenient functions to allow users to easily add custom dimension scales to the data. Dimension scales are described by: **1)** a data array with the scale's data, **2)** the name of the unit of the data values, **3)** a human-readable description of the contents of the scale, **4)** the name of the scale, and **5)** the axis with which the scale is associated. The ability to easily generate custom dimension scales enables users to conveniently associate additional descriptions with the data, e.g., scales describing the classification of electrodes or time values into unique groups/clusters or to encode the occurrence of different events in time, such as, speech events or neural spikes among many others. Dedicated functions for look-up and retrieval of all or select dimensions scales—including all auxiliary data, e.g., the units or description of the scale(s)—ease the integration and use of dimensions scales for analytics.

Dimensions scales are limited in that they are one-dimensional in nature—specifically, even though the scale's dataset may be an arbitrary n-dimensional array, the data is strictly associated with a particular dimension of the main dataset—and are not well-suited to describe complex structures, such as, multi-level data classifications with overlapping clusters. We, therefore, use the */data/internal/ecog_data_#/annotations_# : AnnotationDataGroup* module (see Sec. 3.3.2) for storage and management of complex data annotations. The anatomical data, e.g., is automatically stored both via a dimension scale as well as annotations to facilitate the use of the anatomy in advanced analytics. The *BrainDataECoG* API also provides a number of convenience functions to assist with the interaction with and creation of custom collections of data annotations for the *raw_data*. Annotations play a critical role in advanced analytics based on the classification of the data, e.g., based on the occurrence of events in time such as neural spikes or speech events. The definition of speech events in particular depends heavily on the ability to define many different types of annotations in conjunction with complex user-defined metadata associated with the annotations. For example, speech events occur at a broad range of nested classes, ranging from individual phonemes to syllables, words, and sentences etc.. The same speech event can occur arbitrary often during the course of an experiment—e.g, patient says 'baa'—and different events can overlap—e.g., the sound 'baa' is part of the words 'bad'. The ability to query annotations to locate particular speech events and subsequently analyze the data with such events is critical to the study of neural activity during speech production.

Data annotation provides an ideal framework for storage and analysis of many derived classifications of electrical recordings, for example to define the occurrence of neural spikes via spike sorting.

As described earlier, the creation of managed objects is standardized, i.e., to create a new ***BrainDataECoG*** managed object we simply call the ***BrainDataECoG.create(...)*** function. All required data structures are initialized during the creation process, ensuring that the data file is always valid. Other, optional structures (e.g, the anatomy) may be saved directly during the creation or added later. To ease the use of ***BrainDataECoG*** during data acquisition, the create process allows the raw data and associated dimension scales to be initialized as empty datasets. As new recordings are acquired over time the ***raw_data*** and associated dimensions scales are then automatically expanded to accommodate the new data. With this so-called *auto-expand-data* feature enabled we can, for example do the following:

```
>>> from brain.dataformat.brainformat import BrainDataFile, BrainDataECoG
>>> import numpy
>>> brainfile = BrainDataFile.create('testfile.h5')   # Create the file and initialize the data hierarchy
>>> internal_data = brainfile.data().internal()        # Get the managed object for storing internal data
>>> ecog_data = BrainDataECoG.create(parent_object=internal_data ,     # Add to /data/internal
                                     ecog_data_shape=(32,0),          # Empty recording for 32 electrodes
                                     ecog_data_type='f',              # Store float data values
                                     chunks=True)                     # Store the data using chunking
>>> ecog_data.set_auto_expand(True)                                    # Enable auto expansion
>>> ecog_data[:, 0:1000] = numpy.arange(32*1000).reshape(32,1000)      # Add new data
```

Note when adding the new data, the shape of our ECoG dataset is automatically expanded to 32×1000 and all one-dimensional dimension-scales that are associated with the time axis are automatically expanded to match the new data shape so that we can also conveniently update the data of dimension scales without having to resize the datasets manually. As the above example illustrates, the ***BrainDataECoG*** API provides a convenient interface that allows us to directly interact with the primary ***raw_data*** via array slicing while auxiliary data, e.g., the sampling rate, layout, annotations etc., can be easily retrieved via corresponding access functions or key-based slicing(similar to Python dictionaries).

4.1.3 Storing Processed ECoG Data

In practice, ECoG and other temporal voltage recordings across multiple sensors, are often further processed to extract specific, fixed-length tokens/features (e.g, phonemes or task trials) from the data. As a result the data is often reorganized as a three-dimensional array of $space \times time \times token$. The ***BrainDataECoGProcessed*** module is derived from ***BrainDataECoG*** and extends it to support storage of such processed data. Specifically: **1)** the primary dataset is extended by a third dimension to store the different $channels$ and the dataset is renamed to ***processed_data***, **2)** a set of new optional dimension scales are specified to describe the $frequency\_bands$, $token\_id$, and $token\_name$. Similar to the anatomy data, token data is stored both as dimension scales as well as via metadata-rich, searchable annotations to facilitate data analysis.

Figure 9 shows an example visualization of a processed ECoG dataset stored using our proposed data format. The tree view on the left shows the file structure, including all datasets associated with the ***/data/internal/ecog_data_processed_#*** group. The table view on the right then shows the contents of the primary ***processed_data*** dataset and the curve plot shows the voltage signal over time for a select set of tokens/electrodes. The properties view at the bottom then shows the shape, data type, and attributes associated with the main dataset.

5 CONCLUSIONS AND FUTURE WORK

Neuroscience is facing an incredible big data challenge. Efficient and easy-to-use data standards are a critical foundation to solving this challenge by enabling efficient storage, management, sharing, and analysis of complex neuroscience data. Standardizing neuroscience data is as much about defining common schema and ontologies for organizing and communicating

data as it is about defining basic storage layouts for specific data types. Arguably, the focus of a neuroscience-oriented data standard should be on addressing the application-centric needs of organizing scientific data and metadata, rather than on reinventing file storage and format methods. For the development of BRAINformat we have used HDF5 as the basic storage format, because it already satisfies a broad range of the more basic format requirements.

The complexity and variety of experiments and the diversity of data types and acquisition modalities used in neuroscience make the creation of a general, all-encompassing data standard a daunting—if not futile—task. We have introduced the concept of *Managed Objects* (and *Managed Types*), which—in combination with an easy-to-use, formal format-specification document standard and API—enables us to divide & conquer the data standardization problem in a modular and extensible fashion. Format components specified using these concepts can be easily reused and extended and the format-compliance of file objects can be easily verified using the the BRAINformat library. The format specification API and managed object API implemented in BRAINformat are not specific to neuroscience, but define application-independent design concepts that enable us to efficiently create application-oriented data format modules. Based on these concepts we have developed an extensible data standard for neuroscience data that is portable, scalable, extensible, self-describing, and that supports self-contained (single-file) and modular (multiple-linked-files) storage.

We have also introduced a novel data format module and API for storage and management of advanced data annotations, enabling scientists to further characterize and organize data subsets via additional metadata descriptions. Additionally, we described the novel concept of relationship attributes for modeling and use of structural and semantic relationships between primary storage objects—including advanced index map relationships based on the concept of relationship chains. Although these features are available through an API, the data stored in the format is fully specified and human readable, so that domain scientists can access the data even without our API. These advanced capabilities fill critical gaps in the portfolio of available tools for creating advanced data standards for modern scientific data. The BRAINformat library is open source, has detailed developer documentation and user tutorials, and is freely available at: `https://bitbucket.org/oruebel/brainformat`.

In our future work we plan to extend the BRAINformat via advanced support for metadata ontology and data type specification capabilities and efficient metadata search, as well as expansion of the data annotation modules by supporting additional data selection schema and representations. We will develop capabilities to enable linking and interaction with external data stored in third-party formats (e.g. movies or images) and will develop additional data modules needed to provide a broader coverage of use cases in neuroscience research.

For concreteness, so far we have focused application of BRAINformat to electrocorticography data collected from neurosurgical patients during speech production. At the surface, it may appear that this is a specialization that hinders the general applicability of our work to the broader neuroscience community. However, the electrocorticography data shares many requirements with standard electrophysiology data collected by the community: storage of voltage recordings over time across multiple spatial distributed sensors with heterogenous geometries, complex and multi-tiered task descriptions, post-hoc processing of raw data to extract the signal of interest, the association of physiology data with multi-modal data streams collected simultaneously by other devices, the linkage of data associated with the same 'task' across multiple sessions, and the necessity to store rich meta-data to make sense of it all. Use of our format as input to popular spike-sorting algorithms, such as KlustaKwik [7], should be straightforward. Importantly, the utilization of metadata-rich Annotations are a natural way to encode the occurrence of spikes and associated parameters in the context of the original data, while relationship attributes provide an ideal foundation for recording relationships between analytics and other data. Therefore, application of BRAINformat to physiology data collected in model species during standard sensory, motor, and cognitive neuroscience tasks should be straightforward. Indeed, this has been our goal all along.

ACKNOWLEDGMENTS

This work was supported by Laboratory Directed Research and Development (LDRD) funding from Berkeley Lab, provided by the Director, Office of Science, of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231. This research used resources of the National Energy Research Scientific Computing Center, a DOE Office of Science User Facility supported by the Office of Science of the U.S. Department of Energy under Contract No. DE-AC02-05CH11231. We would like to thank Fritz Sommer, Jeff Teeters, Annette Greiner for helpful discussions. We would like to thank the members of the Chang Lab (UCSF) and Denes Lab (LBNL) for helpful discussions, data, and support.

LEGAL DISCLAIMER

REFERENCES

[1]JSON: JavaScript Object Notation, 1999 – 2015. [ONLINE] `http://json.org/`.

[2]T. Bray, J. Paoli, C. Sperberg-McQueen, E. Maler, and F. Yergeau. Extensible markup language (xml), 2008. [ONLINE] `http://www.w3.org/TR/2008/REC-xml-20081126/`.

[3]J. Clarke and E. Mark. Enhancements to the extensible data model and format (xdmf). In *DoD High Performance Computing Modernization Program Users Group Conference, 2007*, pages 322–327, June 2007.

[4]P. Gleeson, A. Crook, R. C. Cannon, M. L. Hines, C. O. Billings, M. Farinella, T. M. Morse, A. P. Davison, S. Ray, U. S. Bhalla, S. R. Barnes, Y. D. Dimitrova, and R. A. Silver. NeuroML: A Language for Describing Data Driven Models of Neurons and Networks with a High Degree of Biological Detail. *PLoS Computational Biology*, 6(6), 2010.

[5]J. Grewe, T. Wachtler, and J. Benda. A Bottom-up Approach to Data Annotation in Neurophysiology. *Frontiers in Neuroinformatics*, 5(16), 2011.

[6]S. N. Kadir, D. F. M. Goodman, and K. D. Harris. Klustakwik, 2013 – 2015. [ONLINE] `http://klusta-team.github.io/klustakwik/`.

[7]S. N. Kadir, D. F. M. Goodman, and K. D. Harris. High-dimensional cluster analysis with the Masked EM Algorithm. *arXiv.org*, September 2013. [arXiv:1309.2848 [q-bio.QM]].

[8]P. Klosowski, M. Koennecke, J. Tischler, and R. Osborn. Nexus: A common format for the exchange of neutron and synchroton data. *Physica B: Condensed Matter*, 241:151–153, 1997.

[9]F. R. Maia. The coherent x-ray imaging data bank. *Nature methods*, 9(9):854–855, 2012.

[10]R. Rew and G. Davis. NetCDF: an interface for scientific data access. *Computer Graphics and Applications, IEEE*, 10(4):76–82, July 1990.

[11]O. Rübel, A. Greiner, S. Cholia, K. Louie, E. W. Bethel, T. R. Northen, and B. P. Bowen. Openmsi: A high-performance web-based platform for mass spectrometry imaging. *Analytical Chemistry*, 85(21):10354–10361, 2013.

678 [12]S. Shasharina, J. R. Cary, S. Veitzer, P. Hamill, S. Kruger, M. Durant, and D. A. Alexander. VizSchema–Visualization
679 Interface for Scientific Data. In *IADIS International Conference, Computer Graphics, Visualization, Computer Vision and*
680 *Image Processing*, page 49, 2009.
681 [13]A. Stoewer, C. J. Kellner, and J. Grewe. NIX, 2014. [ONLINE] `https://github.com/G-Node/nix/wiki`.
682 [14]The HDF Group. Hierarchical Data Format, version 5, 1997-2015. [ONLINE] `http://www.hdfgroup.org/HDF5/`.
683 [15]The HDF Group. HDFView, 2006 – 2015. [ONLINE] `http://www.hdfgroup.org/products/java/`
684 `hdfview/`.
685 [16]U.S. Army Research Laboratory. eXtensible Data Model and Format (XDMF), 2011 – 2015. [ONLINE] `http://www.`
686 `xdmf.org`.
687 [17]K. Wu, E. J. Otoo, and A. Shoshani. An efficient compression scheme for bitmap indices. *Lawrence Berkeley National*
688 *Laboratory*, 2004.
689 [18]K. Wu, E. J. Otoo, and A. Shoshani. Optimizing bitmap indices with efficient compression. *ACM Transactions on Database*
690 *Systems (TODS)*, 31(1):1–38, 2006.

Supplementary Material: BRAINformat: A Data Standardization Framework for Neuroscience Data

Oliver Rübel*, Prabhat, Peter Denes, David Conant, Edward Chang, and Kristofer Bouchard

*Correspondence:
Author Name: Oliver Rübel
oruebell@lbl.gov

CONTENTS

1 SUPPLEMENT: TUTORIAL ON RELATIONSHIP ATTRIBUTES

Brief Introduction to Relationship Attributes

Oliver Rübel

August 11, 2015

Contents

1 Basic Setup

This section contains code for the basic setup of this tutorial.

```
In [1]: # Import basic packages needed for this notebook
        import sys
        import numpy as np
        from scipy.ndimage import zoom as image_zoom
        import h5py
        from IPython.display import Image
        from tempfile import NamedTemporaryFile
        try:
            %matplotlib inline
            import matplotlib
            import matplotlib.pyplot as plt
            HAS_PLT = True
        except:
```

```
    HAS_PLT = False

# If the brain lib is not installed in your default path then set the path to it
sys.path.append("/Users/oruebel/Devel/BrainFormat")

# Import the classes related to the management of relationships
from brain.dataformat.spec import RelationshipTargetSpec, RelationshipSpec, BaseSpec
from brain.dataformat.base import RelationshipAttribute
```

```
In [2]: # Create a temporaryr HDF5 file for testing
        tempfile = NamedTemporaryFile()
        test_file = h5py.File(tempfile.name, 'a')
```

```
In [3]: # Create some simple example datasets in our temporary file for testing
        test_file['t1'] = np.arange(10)
        test_file['t2'] = np.arange(10) + 10
        test_file['t3'] = np.arange(10) + 5.1
        test_file['t2d_1'] = np.arange(100).reshape((10,10))
        test_file['t2d_2'] = np.arange(100).reshape((10,10)) + 10
        # Assign the datasets to variables
        t1 = test_file['t1']
        t2 = test_file['t2']
        t3 = test_file['t3']
        t2d_1 = test_file['t2d_1']
        t2d_2 = test_file['t2d_2']
```

2 Overview

The concept of relationships describes an extension of the HDF5 primitives to enable us to describe relationships between objects (i.e., Groups and Datasets) in HDF5 files. The description of relationships is based on two basic concepts: 1) the formal specification of relationships and 2) a standard for storing relationship specifications as HDF5 attributes.

Specification Relationships are described in Python via JSON serializable dictionaries that describe the `source`, `target`, `type`, `description`, and storage properties of the relationship (see below for details). The BRAINformat library provides the following main classes to help with the specification of relationships:

- `RelationshipTargetSpec :` Define a dictionary describing the `target` of the relationship
- `RelationshipSpec :` Define a dictionary with the complete specification of the relationship

Storage Relationships are described via a formal specification and stored in HDF5 as attributes. The BRAIN library prepends `RELATIONSHIP_ATTR` to the attribute name to help with the identification of attributes that define relationships. The specification of relationships is typically stored as a JSON string in HDF5. The use of JSON is primarily a choice of convenience. Alternatively, one could also use an HDF5 compound dataset to store relationship specifications. While the use of a compound dataset has advantages—mainly that it relies only on HDF5 primitives—and is not hard to do in Python, the use of changing compound datatypes can be tricky in other (typed) languages. The BRAINformat library provides the following main classes to help with the interaction with relationships stored in HDF5:

- `RelationshipAttribute :` Store, retrieve, and resolve relationships described in HDF5

2.1 Creating a new RelationshipAttribute

In this example we will create a relationship between t1 and t2 that says that the objects in t1 are ordered the same way as the objects in t2.

A relationship consists of a source, target, and definition of the relationship. The source of a relationship is defined simply by the dataset (or group) that the Relationship Attribute (that describes the relationship) is assigned to. We may define an optional `axis` to describe, which dimension the relationship is associated with.

The target is defined via a `RelationshipTargetSpec` specification, describing how we can get from the source to the target object.

The actual function of the relationship is then mainly defined by the relationship type.

2.1.1 Creating a Relationship: Step-by-Step

Specifying the target via a RelationshipTargetSpec First, we need to create a description of the target we are pointing to, i.e., how do we get from `t1` to `t2`. Since we know `t1` and `t2`, we can use the convenient function `from_objects(...)` provided by the `RelationshipTargetSpec` class to easily create this description.

```
In [4]: target_spec_t1_to_t2 = RelationshipTargetSpec.from_objects(source_object=t1,
                                                                   target_object=t2)
```

Let's see what the description of the target specification looks like. Since RelationshipTargetSpec inherits from BaseSpec we can easily convert the specificaiton to/from JSON.

```
In [5]: print target_spec_t1_to_t2.to_json(pretty=True)
```

```
{
    "axis": null,
    "dataset": "t2",
    "filename": null,
    "global_path": null,
    "group": null,
    "prefix": null
}
```

`t2` and `t1` are both located in the same parent group so the description simply contains the name of `t2`. We could have naturally also created the same specification ourselves via `RelationshipTargetSpec(dataset='t2', group=None, prefix=None)` but if we know the HDF5 objects we want to relate, then using `from_objects(...)` is usally simpler.

Specifying a relationship via a RelationshipSpec Second, now that we have a description of our target we can go ahead and specify our relationship.

```
In [6]: rel_spec_t1_to_t2 = RelationshipSpec(attribute='rel_t2',
                                             target=target_spec_t1_to_t2,
                                             relationship_type='order',
                                             description='Test relationship')
        # attribute : Name of the attribute we want to use to store the relationship
        # target : The specification of our target
        # relationship_type : How are the datasets related with each other
        # description : Human-readable description of the relationship
```

Saving a relationship via a RelationshipAttribute Finally, we need to save our relationship to file by creating a new RelationshipAttribute. As usual, we provide a create function for this purpose.

```
In [7]: rel1 = RelationshipAttribute.create(
            parent_object=t1, # The source of the relationship
            relationship=rel_spec_t1_to_t2)  # The specification  of the relationship
```

To confirm that we indeed now have a new attribute on `t1`, let's have a look at all attributes:

```
In [8]: for attr_name, attr_value in t1.attrs.iteritems():
            print attr_name
            print BaseSpec.from_json(attr_value).to_json(pretty=True)  # Just to prettyfy the print
            print ""
```

```
RELATIONSHIP_ATTR_rel_t2
{
    "attribute": "rel_t2",
    "axis": null,
    "description": "Test relationship",
    "optional": false,
    "prefix": null,
    "properties": null,
    "relationship_type": "order",
    "target": {
        "axis": null,
        "dataset": "t2",
        "filename": null,
        "global_path": null,
        "group": null,
        "prefix": null
    }
}
```

As we can see, we now have a new relationship attribute. The BRAIN library prepended the `RELATIONSHIP_ATTR` to help with the identification of attributes that define relationships. The descriptions is typically stored as JSON. One could also use an HDF5 compound dataset to describe a relationship. While the use of a compound dataset has many advantages—we stay in HDF5, more compact etc.—and is not hard to do in Python, the use of changing compound datatypes can be tricky in other languages.

2.1.2 Creating a Relationship: The Shortcut

Instead of constructing the specification of the target and the relationship and then crating the `RelationshipAttribute`, we can use the `RelationshipAttribute.create(...)` function to conveniently create everything in one shot:

```
In [9]: rel_temp = RelationshipAttribute.create(parent_object=t1,
                                                target_object=t2,
                                                attribute='rel_temp_to_t2',
                                                relationship_type='order',
                                                description='Test relationship')
        print rel_temp.relationship_spec.to_json(pretty=True)  # Print for validation purposes
```

```
{
    "attribute": "rel_temp_to_t2",
    "axis": null,
    "description": "Test relationship",
    "optional": false,
    "prefix": null,
    "properties": null,
    "relationship_type": "order",
    "target": {
        "axis": null,
        "dataset": "t2",
```

```
        "filename": null,
        "global_path": null,
        "group": null,
        "prefix": null
    }
}
```

2.2 Using Relationships

We can use the RelationshipAttribute class (and instances thereof) to directly interact with relationships.

```
In [10]: source = rel1.source              # Get the source of the relationship (here, t1)
         source_axis = rel1.source_axis    # Get the axis to which the relationship applies
         target = rel1.target              # Get the target of the relationship (here, t2)
         target_axis = rel1.target_axis    # Get the axis on the target of the relationship
         rtype = rel1.relationship_type    # Get the type of the relationship
         rspec = rel1.relationship_spec    # Get RelationshipSpec with relationship's specification
         tspec = rel1.target_spec          # Get RelationshipTargetSpec with the sepc of the target
                                           # Alternatively we could also get the target spec from
                                           # our RelationshipSpec via:
                                           # >>>  tspec = rspec['target']

         print "(Source, Axis) = (" + str(source) + ", " + str(source_axis) + ")"
         print "(Target, Axis) = (" + str(target) + ", " + str(target_axis) + ")"
         print "Relationship Type = " + str(rtype)
         # We here call to_json(True) simply to make the output easier to read
         print "Relationship Specification " + rspec.to_json(pretty=True)
```

```
(Source, Axis) = (<HDF5 dataset "t1": shape (10,), type "<i8">, None)
(Target, Axis) = (<HDF5 dataset "t2": shape (10,), type "<i8">, None)
Relationship Type = order
Relationship Specification {
    "attribute": "rel_t2",
    "axis": null,
    "description": "Test relationship",
    "optional": false,
    "prefix": null,
    "properties": null,
    "relationship_type": "order",
    "target": {
        "axis": null,
        "dataset": "t2",
        "filename": null,
        "global_path": null,
        "group": null,
        "prefix": null
    }
}
```

2.2.1 Relationship Slicing

For many standard relationship types, the `RelationshipAttribute` API also supports slicing to map selections defined on the `source` to the `target`. The `RelationshipAttribute` API maps the selection but it is up to the user to apply the selection to the `target` to load data if desired. When loading data from the the `target` we need to consider the `target_axis` if set.

```
In [11]: # Loading data from the relationship's source, i.e, t1
         temp_data = rel1.source[0:3]
         print "Loading t1[0:3]: " + str(temp_data)

         # We can map slicing operations from the source to the
         # target by slicing into the relationship
         temp_data = rel1[0:3]
         print "Mapping [0:3] to the target: " + str(temp_data)

         # We can now use the mapped slice to directly access
         # data in the target, i.e., t2
         temp_data = rel1.target[rel1[0:3]]
         print "Loading t2[0:3] via the relationship: " + str(temp_data)
```

```
Loading t1[0:3]: [0 1 2]
Mapping [0:3] to the target: slice(0, 3, None)
Loading t2[0:3] via the relationship: [10 11 12]
```

3 Types of relationships and their uses

3.1 `Order`

Type: `order`

Description:

Typical use case:

Slicing: `order` selections datasets can be directly mapped between two datasets without the need for complex data processing, i.e., the selection given for the `source` is identical to what we need to do for the `target`. The `RelationshipAttribute` API supports this mapping directly. `order` relationship are usually used between datasets since objects in a Group (i.e., HDF5 group) don't have a predefined order. The `RelationshipAttribute` API supports `order` relationships between Groups by assuming that keys in the groups are sorted alphabetically and maps selections (typically names of objects or indexes/slices into the ordered list of keys) accordingly. The `RelationshipAttribute` API currently does not support mapping `order` selections between datasets and groups (which are expected to be not very common).

Example

See example above

3.2 `Equivilant`

Type: `equivalent`

Description: In addition to order, this relationship type expresses that the source and target object encode the same data (even if they might store different values). This relationship also implies that the source and target contain the same number of values ordered in the same fashion.

Typical use case: Any time the same data is stored multiple times with different encodings, e.g., a user may store a dataset of strings with the names of tokens and store another dataset with the integer ID of the tokens. This is often useful to ease data processing, e.g., while IDs may be better suited for data processing, string names may be better suited for human interpretation.

Slicing: Same as for `order`.

Example See `order` example above

3.3 `Shared Encoding`

Type: `shared_encoding`

Description: The `target` and `source` contain values with the same encoding, i.e., values can be directly compared. The specification of a `target_axis` usually does not make sense for this type of relationship as this relationship refers directly to the values of datasets not their axes.

Typical use case: Generally any time two objects (datasets or groups) contain data with the same encoding, i.e., values or names that can be directly compared via `==` (and $<$, $>$ in the case of datasets). Imaging, e.g, the case where we have two datasets with dimension scales that demark regions where a patient performed a particular task.

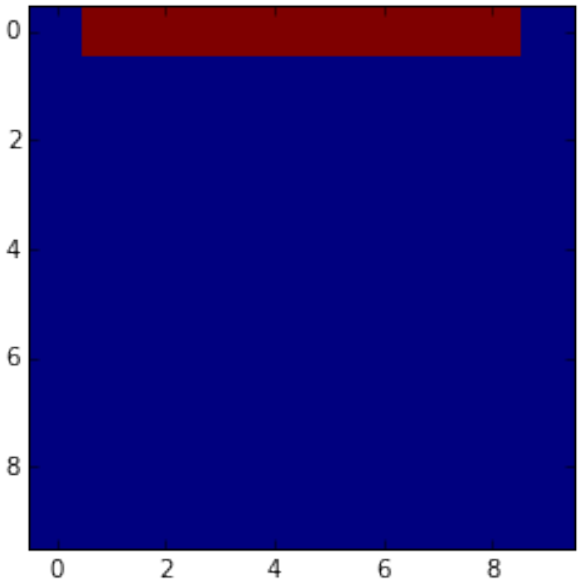
Slicing: The `RelationshipAttribute` API supports slicing for `shared_encoding` relationships as follows. We assume that we want to locate the same values int `target` as in the `source`. As such we will load the requested data from the `source` and then locate all values in `target` that are equal to at least one of the loaded values.

Example: In the below example we create an example `shared encoding` relationship between the 1D dataset `t1` and the 2D dataset `t2d_1`. As we can see the slice defined on the source is translated to a boolean map indicating the corresponding values in the `target` dataset.

```
In [12]: rel_se_1 = RelationshipAttribute.create(parent_object=t1,
                                                 target_object=t2d_1,
                                                 attribute='rel_t2d_1',
                                                 relationship_type='shared_encoding',
                                                 description='test relationship')
```

```
In [13]: print 'Selected data in the source dataset: ' + str(rel_se_1.source[1:9])
         mapped_selection = rel_se_1[1:9]
         print 'Selected data in the target dataset: ' + str(rel_se_1.target[mapped_selection])
         print 'Mapping the selection [1:9] from the 1D source to the 2D target'
         if HAS_PLT:
             plt.imshow(mapped_selection, interpolation='nearest')
         else:
             print mapped_selection
```

```
Selected data in the source dataset: [1 2 3 4 5 6 7 8]
Selected data in the target dataset: [1 2 3 4 5 6 7 8]
Mapping the selection [1:9] from the 1D source to the 2D target
```



3.4 Shared Ascending Encoding

Type: `shared_ascending_encoding`

Description: The `target` and `source` contain values with the same encoding, i.e., values in the two datasetscan be directly compared. In addition, values are expected to be in ascending order. An example would be a 1D time dimension for two different datasets that are synchronized to a global clock. The specification of a `target_axis` usually does not make sense for this type of relationship as this relationship refers directly to the values of datasets not their axes.

Typical use case: Relate ordered dimensions, e.g., time axis

Slicing: The `RelationshipAttribute` API supports slicing for `shared_ascending_encoding` as follows:

1. In the general case, the shared_ascending_encoding behaves exactly like the the shared_encoding
2. In the special case, however, where i) the source and target are 1D datasets and ii) a pure range selection is performed (i.e., a single slice with just a start and stop but no step) the range of the slice is mapped from the source to the target (rather than equals). This is useful, e.g., in the case where we have a time axis and we want to locate a time-frame rather than exact matching values between two datasets. NOTE: If we want to ensure that behavior 1. is always used, then we can simply define a step for slice, i.e., instead of [0:10] we can instead write [0:10:1].

Example:

```
In [14]: # Here we are creating a new relationship between t1 and t3
         # with the type shared_ascending_encoding
         rel2 = RelationshipAttribute.create(parent_object=t1,
                                             target_object=t3,
                                             attribute='rel_t3',
                                             relationship_type='shared_ascending_encoding',
                                             description='Test relationship')
```

```
In [15]: # We next here show how we can use the relationship to slice into the data
         print "Source data all: " + str(rel2.source[:])
         print "Target data all: " + str(rel2.target[:])
         print "Source selecton: " + str(np.s_[2:9])
         print "Selected source data: " + str(rel2.source[2:9])
         print "Mapped seleciton: " + str(rel2[2:9])
         print "Selected target data: " + str(rel2.target[rel2[2:9]])
```

```
Source data all: [0 1 2 3 4 5 6 7 8 9]
Target data all: [  5.1   6.1   7.1   8.1   9.1  10.1  11.1  12.1  13.1  14.1]
Source selecton: slice(2, 9, None)
Selected source data: [2 3 4 5 6 7 8]
Mapped seleciton: slice(0, 3, None)
Selected target data: [ 5.1  6.1  7.1]
```

In the above example we selected the range `[2:9]` from dataset `t1` (i.e. the source), which are the values `[2 3 4 5 6 7 8]`. The target dataset t3, however, contains only the values `[5.1 6.1 7.1 8.1 9.1 10.1 11.1 12.1 13.1 14.1]`. When mapping the value range (2 to 8) to the target we, hence, retrieve less values, specifically `[5.1 6.1 7.1]`.

As mentioned above, we can force to use an equals comparison when mapping the selection by defining the step parameter of the slice, which in this case will result in no values being selected as none of the values match exactly:

```
In [16]: # Now lets see what happens if don't sl
         mapped_selection = rel2[2:9:1]
         print "Mapped selection (using equals): " + str(mapped_selection)
         print "Total values selected in target: " + str(mapped_selection.sum())
```

```
Mapped selection (using equals): [False False False False False False False False False False]
Total values selected in target: 0
```

3.5 Indexes

Type: `indexes`

Description: The source dataset contains indices into the `target` dataset. These are often integer indices, however, e.g, if the relationship points to a Group, then the `source` dataset may also contain strings selecting the objects stored in the `Group`. The source of such a relationship, however, should always be a dataset.

Typical use case: A typical use is to describe basic data structures, e.g., we may store a list of unique tokens and a larger array that stores integer indicies into the list of tokens.

Slicing: The `RelationshipAttribute` API maps slicing operations by retrieving the indices from the source dataset and returing a list of the indicies. NOTE: in case that the `target` is a Group, we may need to iterate over the returned selection as h5py.Group objects do not support simultaneous selection of multiple objects.

1D Example

```
In [17]: # First, let's create some new datasets
         test_file['token_names'] = np.asarray(['aah', 'bee', 'cat', 'bat', 'fat'])
         test_file['token_ids'] = np.random.randint(0, 5, 20)   # Indicies into the token_names array
         token_names = test_file['token_names']
         token_ids = test_file['token_ids']

In [18]: # Here just a quick example of how one would use this sort of strucutre in practice
         print "Token IDs: " + str(token_ids[:])
         print "Mapped Token IDs: " + str(token_names[:][token_ids[:]])

Token IDs: [1 2 3 0 3 0 2 2 4 1 2 3 0 4 1 0 3 4 2 4]
Mapped Token IDs: ['bee' 'cat' 'bat' 'aah' 'bat' 'aah' 'cat' 'cat' 'fat' 'bee' 'cat' 'bat'
 'aah' 'fat' 'bee' 'aah' 'bat' 'fat' 'cat' 'fat']

In [19]: # To make the relationship between the datasets explicit we create a
         # RelationshipAttribute, which describes that token_ids indexes token_names
         rel_i_1 = RelationshipAttribute.create(parent_object=token_ids,
                                                target_object=token_names,
                                                attribute='rel_index_target',
                                                relationship_type='indexes',
                                                description='token_ids indexes token_names')

In [20]:     # Now we can use the relationship to transparently perform the mapping from
             # the token_ids arrays to the token_names without having to know the datasets
             target_data = rel_i_1.target[:]     # Here we load the target data into memory
                                                 # because h5py.Dataset does not slicing with
                                                 # out of order index lists, but numpy does
         print target_data[rel_i_1[10:20]]

['cat' 'bat' 'aah' 'fat' 'bee' 'aah' 'bat' 'fat' 'cat' 'fat']
```

Exampled: 2D

```
In [21]: # In this example we will show how we can do something similar but where
         # the target is a multi-dimensional array
         test_file['matrix_data'] = np.arange(100).reshape(10,10)
         temp_index = np.random.randint(0, 5, size=(2,20))
         test_file['matrix_index'] = temp_index   # Indicies into the matrix_data dateaset
         matrix_data = test_file['matrix_data']
         matrix_index = test_file['matrix_index']
```

```
In [22]: # To make the relationship between the datasets explicit we create a
         # RelationshipAttribute, which describes that matrix_index indexes matrix_data
         rel_i_2 = RelationshipAttribute.create(parent_object=matrix_index,
                                                target_object=matrix_data,
                                                attribute='rel_index_target_2D',
                                                relationship_type='indexes',
                                                axis=0,
                                                description='matrix_index indexes matrix_data')
```

Alternatively we could also set to:
`axis={'INDEXING_AXIS':0, 'STACK_AXIS':None}`
instead of `axis=0`. In either case the above indicates that we have multi-dimensional indicies where the first dimenions is used to store the two-dimenionsal indicies, while the remaining dimenions are the intrinsic dimenions of the matrix_index dataset itself (whatever they may be).

```
In [23]: # Mapping the selection from matrix_index to the matrix_data
         mapped_selection = rel_i_2[1:10]
         # Printing the resulats of the mapping
         print "Mapped Selection: (shape=" + str(mapped_selection.shape) + ")"
         print mapped_selection
         # Applying the multi-dimensional selection
         selected_matrix_data = matrix_data[:][ mapped_selection[0,:] , mapped_selection[1,:] ]
         # h5py.Dataset does not support complex multi-dimensional selection but numpy does.
         # For convenience we load the full matrix_data[:] first and then apply the selection
         # but we could also iterate over the selection to load one element at-a-time.
         print "Selected Data:"
         print selected_matrix_data
```

```
Mapped Selection: (shape=(2, 9))
[[4 4 2 2 0 2 2 3 1]
 [1 0 3 3 2 1 1 1 0]]
Selected Data:
[41 40 23 23  2 21 21 31 10]
```

In the above example we used an index dataset `temp_index` of shape `(2,20)`, i.e, we ordered the indicies we select along the 2nd (i.e., last dimension). Depending on preference we may also order selections along the first dimension, in which case the data load would have changed to:
`selected_matrix_data = matrix_data[:][ mapped_selection[...,0] , mapped_selection[...,1] ]`

We can determine which notation we need to use by looking at the value of `rel_i_2.source_axis`, which in the above example is `0` — i.e., `rel_i_2.source_axis` identifies the first axis of the source dataset as the axis that describes the components of the indices, which we can easily check:

```
In [24]: rel_i_2.source_axis == 0
```

```
Out[24]: True
```

3.6 Indexes Values

Type: `indexes_values`
Description: The `source` selects certain parts of the `target` based on the values (or keys in case of group(s)) in the `target`. Specification of an axis for the `target` usually does not make sense for this type of relationship. The `indexes_values` relationship implies that the datasets use a `shared_encoding` (see above) and is effectively a special type of `shared_encoding` relationship that beyond the encoding describes that the `source` is selecting data in the `target` based on value.
Typical use case: Value-based referencing of data
Slicing: The `RelationshipAttribute` API supports slicing for `indexes_values` relationships as follows:

- **Source/Target are datasets:** Map the given `selection` via: `target[:] == self.source[selection]`
- **Source is dataset and target is group:** Retrieve the selected keys from the source via: `self.source[source_selection]`
- **Source/Target arge groups:** Both groups contain objects (groups/datasets) with the same names. The mapping of a given `selection` is, hence, trivial and is the `selection` itself.
- **Source is group and target is dataset:** Map the given selection via: `self.target[:] == selection`

3.7 `User`

Type: `user`

Description: Arbitrary user-defined relationship.

Typical use case: Any time a user needs to express a semantic relationship that does not match any of the other supported relationship types.

Slicing: No particular slicing support is implemented for this type.

4 Relationship Chains

Describing relationships between individual objects can be used as a tool to describe more complex relationships between data objects, e.g., relationships that are based on complex mappings between data objects. In practice we can describe such relationships via a chain of relationships. A common type of such a complex chain relationship are what we here refer to as **index map relationships**.

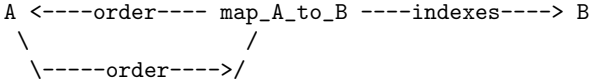
4.1 Index Map Relationships

Lets look at the following example. Imagine we have taken two different images `A` and `B` of a particualr brain at different resolutions or even from different imaging modalities. We now want to analyze our two images in context of each other. This mean when accessing a set of pixels in image `A` we now want to access the corresponding pixels in image `B` (and vice versa). This seamingly simple task, however, is in practice highly complex. Even if our images are perfectly aligned, a user still has to know exactly: i) how the two datasets are related, ii) how to utilize the information from the registration process to map between `A` and `B`, and iii) write complex code to access the data.

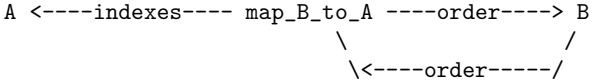
Using relationships we can i) make this process explicit by describing the relationship between `A` and `B` and ii) greatly simplify the process for interacting with the data. Rather than describing the relationship between `A` and `B` directly we can create a map dataset `map_A_to_B`, which stores for each pixel of `A` the corresponding `(x,y)` pixel index in `B`. Accordingly we can also create a corresponding map `map_B_to_A` to model the reverse relationship between `B` and `A`. These maps explicitly describe the relationship between our images `A` and `B` so that a user can directly utilize the mappings without having to perform complex and error-prone index transformation (which would be needed if we described the mapping via scaling, rotation, morphing and other data transformation).

Using `RelationshipAttributes` we can describe this complex relatioship between our images as follows:

- **Mapping between `A` and `B`**



- **Mapping between `B` and `A`**



Here, `A, B, map_A_to_B, map_B_to_A` are datasets and `<--, -->` describe `RelationshipAttributes` of the indicated type. In practice one would often also add a `user` relationship between `A` and `B` to document the specific semantic relationship between the two datasets, e.g:

```
A <----order---- map_A_to_B ----indexes----> B
 \                /                        /
  \-----order---->/                        /
   \                                      /
    \----------------user----------------/
```

While index map relationships may seem complicated, the BRAINformat API provides a series of convient functions, which greatly simplify the us of these kinds of index map relationships in practice; in particular:

- `RelationshipAttribute.create_index_map_relationship(...)` conveniently creates for us all relationship required to define the index map relationshiph
- `RelationshipAttribute.get_index_map_relationship_names(...)` generates a list of all unique names of index map relationships.
- `RelationshipAttribute.get_index_map_relationship(...)` retrieves all relationships that belong to a particular index map relationship.
- `RelationshipAttribute.INDEX_MAP_RELATIONSHIP_POSTFIX` Is a python dictionary describing the standard postfixs used to identify the four relationships used to model the index map relationship, i.e., the mandatory `MAP_TO_TARGET`, `MAP_TO_SOURCE`, `SOURCE_TO_MAP` and to options `SOURCE_TO_TARGET` relationship. The postfix values are used by the API to locate and retrieve index map relationships.
- `RelationshipAttribute.RELATIONSHIP_ATTRIBUTE_PREFIX` As for all Relationship Attributes, this prefix is used to indicate that the different attributes describe regular relationships.
- `RelationshipAttribute.source` and `RelationshipAttribute.target` attributes allow us to easily and transparently locate the `source` and `target` of relationships without having to know the name and location of datasets
- `RelationshipAttribute.__getitem__` allows us to easily perform the mapping of selections using a relationship

4.1.1 Creating Index Map Relationships

Using the functionality described above we can manually create index map type relationships fairly easily, simply by creating the individual `order`, `indexes`, and `user` relationships. To ease and help standardize the creation of such relationships in practice, the `RelationshipAttribute` API provides the convenient shortcut function `RelationshipAttribute.create_index_map_relationship(...)`, which creates all the required relationships specifications and relationship attributes for us.

Create two example image datasets

```
In [25]: # Source Image: 2x2 image where each pixel is the pixel index
         s_x, s_y = 2, 2 # Size of the source image in x and y
         source_image_data = np.arange(s_x * s_y).reshape(s_x, s_y)
         test_file['image1'] = source_image_data
         source_image = test_file['image1']

         # Target Image: 4x4 image scaled up from the source, i.e.,
         # 4 pixel in the target correspond to 1 pixel in the source
         target_image_data = image_zoom(source_image, 2, order=0)
         test_file['image2'] = target_image_data
         target_image = test_file['image2']
```

Create the mappings between `image1` and `image2` Since our target is an up-sampled version of our source image we have for each (x,y) pixel in the source image 4 corresponding pixels in the target image. This means our map must have four dimensions:

1. The first dimension is the `X` dimension of the image
2. The second dimension is the `Y` dimensions of the image
3. The third dimension is used to express the `(x,y)` index of a pixel. We refer to this axis as the `INDEXING_AXIS`.
4. The fourth dimension is used to store the stack if pixels. We refer to this axis as the `STACK_AXIS`.

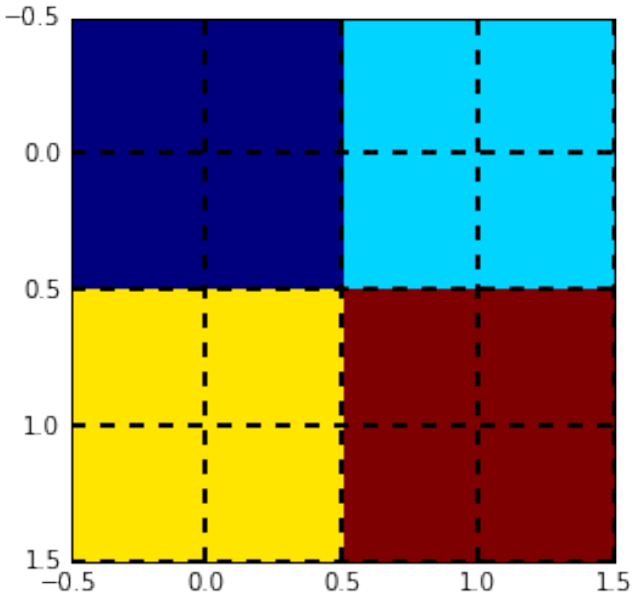
When defining the map, the main restriction is that the intrinsic dimenions of the data (i.e., `X` and `Y`) appear in the same order in the map as they are defined in the `source` dataset. However, we can use arbitrary dimensions for our `INDEXING_AXIS` and `STACK_AXIS`. In practice the `INDEXING_AXIS` and `STACK_AXIS` (if needed) appear either as the first or last dimensions of the map dataset. However, this is not mandatory and we may chose any dimensions as our `INDEXING_AXIS` and `STACK_AXIS` as long a we do not alter the relative ordering or the axis of our `map` dataset relative to our `source` dataset.

```
In [26]: map_source_to_target_data = np.zeros(shape=(s_x, s_y, 2, 4), dtype='uint16')
         for xi in range(s_x):
             for yi in range(s_y):
                 map_source_to_target_data[xi, yi, :, :] = \
                     np.nonzero(target_image_data==source_image_data[xi,yi])
         test_file['map_image1_to_image2'] = map_source_to_target_data
         map_source_to_target = test_file['map_image1_to_image2']

         # Since our images are pixel indicies, the map of the target to source is simply
         # identical to our target image
         test_file['map_image2_to_image1'] = target_image_data
         map_target_to_source = test_file['map_image2_to_image1']
```

Print the image data and maps:

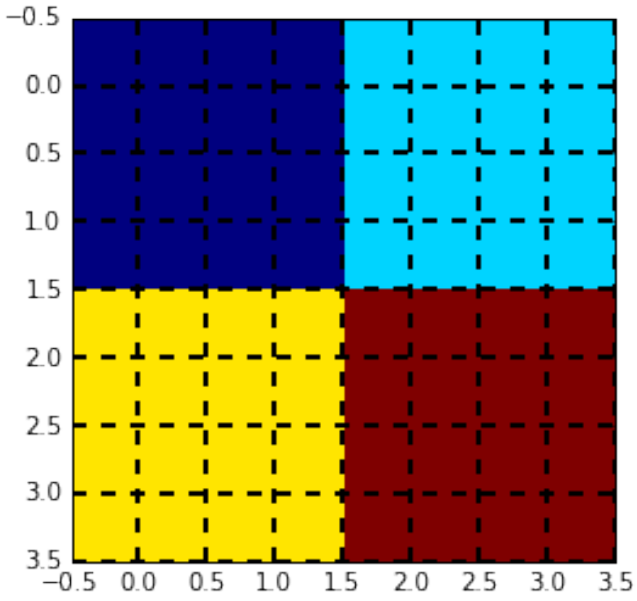
```
In [27]: print "Source Image:"
         if HAS_PLT:
             ax = plt.imshow(source_image_data, interpolation='nearest')
             plt.grid(True, linestyle='--', linewidth=2)
         print source_image_data
```

```
Source Image:
[[0 1]
 [2 3]]
```




```
In [28]: print "Target Image: (this is also the map from the target to the source)"
         if HAS_PLT:
             ax = plt.imshow(target_image_data, interpolation='nearest')
             plt.grid(True, linestyle='--', linewidth=2)
         print target_image_data
```

```
Target Image: (this is also the map from the target to the source)
[[0 0 1 1]
 [0 0 1 1]
 [2 2 3 3]
 [2 2 3 3]]
```




```
In [29]: print "Map of the Source to the Target:"
         print map_source_to_target_data
         print ""
         print "Each pixel in the source maps to four pixels in the target. "
         print "E.g, Pixel (0,0) maps to:"
         print map_source_to_target[0,0,...]
```

```
Map of the Source to the Target:
[[[[0 0 1 1]
   [0 1 0 1]]

  [[0 0 1 1]
   [2 3 2 3]]]


 [[[2 2 3 3]
   [0 1 0 1]]

  [[2 2 3 3]
   [2 3 2 3]]]]

Each pixel in the source maps to four pixels in the target.
E.g, Pixel (0,0) maps to:
[[0 0 1 1]
 [0 1 0 1]]
```

Creating the Index Map Relationship for our Example Image Data

```
In [30]: mapping_relationships = RelationshipAttribute.create_index_map_relationship(
                name='upsampled_image_relationship',
```

```
                map_object=map_source_to_target,
                source_object=source_image,
                target_object=target_image,
                map_indexing_axis=2,  # the 3rd dimension idenifies our indicies
                map_stack_axis=3)     # the 4th dimension has the stack of pixels we relate to
```

That's all. Optionally we could have also specified a `user_description` and/or `use_properties` to also generate a `user` relationship between the two images. We could have also added `mapping_properties` to further describe the mapping we performed in further detail. We here left the `source_axis` and `target_axis` paremeters set to `None` as the relationship refers to the whole source and target datasets.

We can now, naturally, as easily create the index map relationship to also capture the inverse relationship from `image2` to `image1`:

```
In [31]: mapping_relationships_2 = RelationshipAttribute.create_index_map_relationship(
                name='downsampled_image_relationship',
                map_object=map_target_to_source,
                source_object=target_image,
                target_object=source_image)
         # Our map contains direct indicies and has no additional dimension,
         #so we don't need the mp_indexing_axis and mapt_stack_axsi
```

This simple strategy makes the relationship between our images fully explicit so that anyone—without any prior knowledge—can discover and utilize our two images in conjunction. Before we show next, how we can use our relationship to easily interact with our two images, let's have a look at the individual relationships that we just created:

```
   map_image1_to_image2 -----indexes----> image2
```

```
In [32]: print mapping_relationships[0].relationship_spec.to_json(pretty=True)
```


```
{
    "attribute": "upsampled_image_relationship_IMR_MAP_TO_TARGET",
    "axis": {
        "INDEXING_AXIS": 2,
        "STACK_AXIS": 3
    },
    "description": "The source defines a map from /image1to the target of this relationship",
    "optional": false,
    "prefix": null,
    "properties": null,
    "relationship_type": "indexes",
    "target": {
        "axis": null,
        "dataset": "image2",
        "filename": null,
        "global_path": null,
        "group": null,
        "prefix": null
    }
}
```


```
   map_image1_to_image2 -----order----> image1
```

```
In [33]: print mapping_relationships[1].relationship_spec.to_json(pretty=True)
```


```
{
    "attribute": "upsampled_image_relationship_IMR_MAP_TO_SOURCE",
```

```
    "axis": [
        0,
        1
    ],
    "description": "The source defines a map from the target of this relationship to/image2",
    "optional": false,
    "prefix": null,
    "properties": null,
    "relationship_type": "order",
    "target": {
        "axis": null,
        "dataset": "image1",
        "filename": null,
        "global_path": null,
        "group": null,
        "prefix": null
    }
}
```

```
   image1 -----order----> map_image1_to_image2
```

```
In [34]: print mapping_relationships[2].relationship_spec.to_json(pretty=True)
```

```
{
    "attribute": "upsampled_image_relationship_IMR_SOURCE_TO_MAP",
    "axis": null,
    "description": "The target of this relationship defined a mapfrom the source of this relationship t
    "optional": false,
    "prefix": null,
    "properties": null,
    "relationship_type": "order",
    "target": {
        "axis": null,
        "dataset": "map_image1_to_image2",
        "filename": null,
        "global_path": null,
        "group": null,
        "prefix": null
    }
}
```

```
   source -----user----> target
```

```
In [35]: if mapping_relationships[3] is not None:
             print mapping_relationships[3].relationship_spec.to_json(pretty=True)
         else:
             print "No user relationship has been defined"
```

```
No user relationship has been defined
```

4.1.2 Interacting with Index Map Relationships

While index map relationships are quite complex, interacting with index map relationships is in practice fairly easy as we can use the relationships defined on our map dataset to interact with our `source` and `target` datasets. E.g.:

1. We can use our map dataset `map_image1_to_image2` to locate: i) `R1`; the `indexes` relationship to B and ii) `R2`, the order relationship to `A`
2. If we now want to slice into `B` we can use `R1` to select elements in `B`
3. If we want to slice into `A` we can use `R2` to select in `A` (or in many cases, if no axis are specified, we can also just slice into `A` since we have an `order` relationship between `A` and `map_A_to_B`.

Retrieving Index Map Relationships

```
In [36]: #1.1) Locate the map dataset
         map_source_to_target = test_file['map_image1_to_image2']
         #1.2) Locate the indexes relationship from our map to our target (i.e, 'image2')
         r1 = mapping_relationships[0]
         #1.3) Locate the order relationship from our map to our source (i.e, 'image1')
         r2 = mapping_relationships[1]
```

Since we already have our relationships we were able to just access them directly. But the `RelationshipAttribute` API also provides us with a series of convenience functions to help us locate the appropriate attributes, in particular:

- `RelationshipAttribute.get_index_map_relationship_name(...)` : Get a list of all unique names of index map relationships
- `RelationshipAttriute.get_index_map_relationship(...)` : Get a dict of all relationships that define the index map relationship

```
In [37]: # Get the list of names of all index map relationships
         imr_names =  RelationshipAttribute.get_index_map_relationship_names(
             parent_object=map_source_to_target)

         # Get the 'upsampled_image_relationship' relationship
         imr1 = RelationshipAttribute.get_index_map_relationship(
             parent_object=map_source_to_target,
             relationship_name='upsampled_image_relationship')

         # Print the results for validation
         print "Index map relationship names: " + str(imr_names)
         print ""
         print "Index map relationship attributes for 'upsampled_image_relationship':"
         for i, j in imr1.iteritems():
             print (i, j)
```

```
Index map relationship names: [u'upsampled_image_relationship']

Index map relationship attributes for 'upsampled_image_relationship':
('SOURCE_TO_MAP', <brain.dataformat.base.RelationshipAttribute object at 0x106331290>)
('MAP_TO_SOURCE', <brain.dataformat.base.RelationshipAttribute object at 0x106331590>)
('SOURCE_TO_TARGET', None)
('MAP_TO_TARGET', <brain.dataformat.base.RelationshipAttribute object at 0x106331450>)
```

Above we used the map dataset `map_source_to_target` to look up the index map relationship, but we can equally well also retrieve the same information from the `source` dataset `source_image`:

```
In [38]: # Get the list of names of all index map relationships
         imr_names =  RelationshipAttribute.get_index_map_relationship_names(
             parent_object=source_image)

         # Get the 'upsampled_image_relationship' relationship
```

```
imr1 = RelationshipAttribute.get_index_map_relationship(
    parent_object=source_image,
    relationship_name='upsampled_image_relationship')

# Print the results for validation
print "Index map relationship names: " + str(imr_names)
print ""
print "Index map relationship attributes for 'upsampled_image_relationship':"
for i, j in imr1.iteritems():
    print (i, j)
```

```
Index map relationship names: [u'upsampled_image_relationship']

Index map relationship attributes for 'upsampled_image_relationship':
('SOURCE_TO_MAP', <brain.dataformat.base.RelationshipAttribute object at 0x1062208d0>)
('MAP_TO_SOURCE', <brain.dataformat.base.RelationshipAttribute object at 0x1062207d0>)
('SOURCE_TO_TARGET', None)
('MAP_TO_TARGET', <brain.dataformat.base.RelationshipAttribute object at 0x106220890>)
```

In general, relationships are directional, pointing from a particular `source` to a `target`. As such, we can easily look up all relationships with a given object as the `source`. However, unless we explicitly define also inverse relationships—e.g, in addition to `A---indexes--->B` we would also have a relationship `A<-----indexed by-----B`—we cannot directly look up relationships that point to a given object.

In the case of index map relationships this means that we can easily answer the question of "Which objects do I map to?" but answering the question of "Which objects map to me?" is more complex (i.e, we would need to iterate through all possible objects to look up all relationships to see if we are the target).

Using Index Map Relationships to Locate the Source, Target, and Map Now that we have retrieved our index map relationship we can use the relatioship to interact with our data.

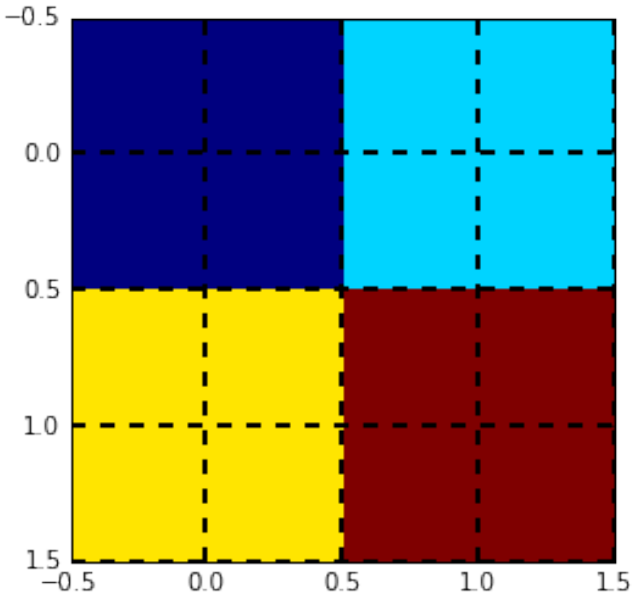
Locate the source and target and map object of our relationship:

```
In [39]: imr1_source = imr1['MAP_TO_SOURCE'].target
         imr1_target = imr1['MAP_TO_TARGET'].target
         imr1_map = imr1['MAP_TO_SOURCE'].source

         # NOTE:
         # `imr1['MAP_TO_SOURCE'].target` is equivalent to `imr1['SOUCE_TO_MAP'].source`
         # `imr1['MAP_TO_TARGET'].target` is equivalent to imr1['SOURCE_TO_TARGET'].target (if exists)
         # `imr1['MAP_TO_SOURCE'].source` is equivalent to  imr1['MAP_TO_TARGET'].source
```

Just to show that we in fact did locate the correct data, lets print it again:

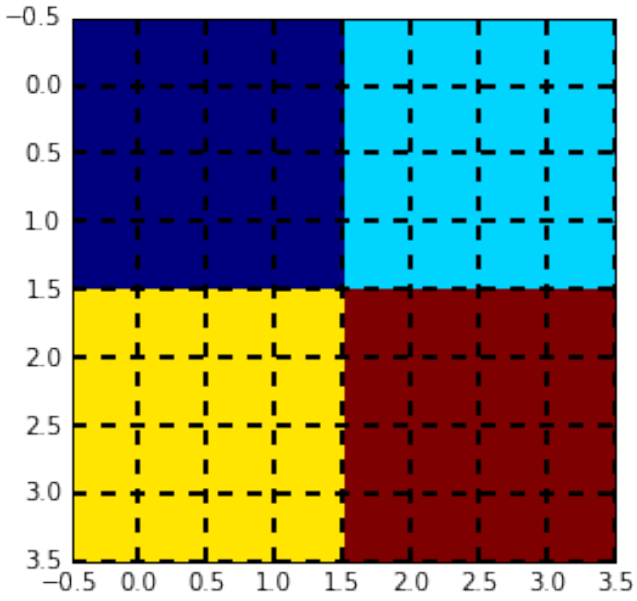
```
In [40]: print "Source Image:"
         if HAS_PLT:
             ax = plt.imshow(source_image_data, interpolation='nearest')
             plt.grid(True, linestyle='--', linewidth=2)
         print source_image_data
```

```
Source Image:
[[0 1]
 [2 3]]
```




```
In [41]: print "Target Image: (this is also the map from the target to the source)"
         if HAS_PLT:
             ax = plt.imshow(target_image_data, interpolation='nearest')
             plt.grid(True, linestyle='--', linewidth=2)
         print target_image_data
```

```
Target Image: (this is also the map from the target to the source)
[[0 0 1 1]
 [0 0 1 1]
 [2 2 3 3]
 [2 2 3 3]]
```




```
In [42]: print "Map of the Source to the Target:"
         print map_source_to_target_data
         print ""
```

```
Map of the Source to the Target:
[[[[0 0 1 1]
   [0 1 0 1]]

  [[0 0 1 1]
   [2 3 2 3]]]


 [[[2 2 3 3]
   [0 1 0 1]]

  [[2 2 3 3]
   [2 3 2 3]]]]
```

When comparing the printed data to what is shown in the Section Creating Index Map Relationships we can see that we retrieved the correct data. Since we used our index map relationship, we did not even need to know names and locations of the datasets.

Using Index Map Relationships to Load Data We can now also use the relationship to directly load data. For the source we can usually directly load the data from the source. However, note, that the index map relationship may refer only to select axis of our dataset. If this is the case, then the `imr1['MAP_TO_SOURCE'].target_axis` as well as `imr1['SOURCE_TO_MAP'].axis` will be set to indicate which axes of the source the relationship applies to.

For the target we can use the `MAP_TO_TARGET` relationship to translate selections from the `source` to the `target`. Similarly, if the relationship refers to only particular axes of the `target` then the `imr1['MAP_TO_TARGET'].target_axis` will be set.

Let's see what happens when we load the first two pixels from the `source` and from the `target`.

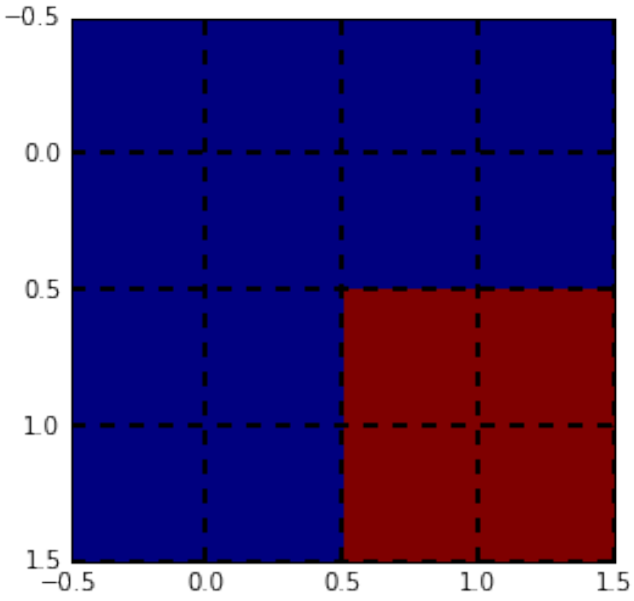
```
In [43]: source_selection = np.s_[1,1]
         target_selection = imr1['MAP_TO_TARGET'][source_selection]
         source_subimage = imr1_source[source_selection]  # We could also just write imr1_source[:,0]
         target_subimage = imr1_target[:][target_selection[...,0,:] , target_selection[...,0,:] ]

         print "Data loaded from source:"
         print source_subimage
         print ""
         print "Data loaded from target:"
         print target_subimage
         print ""
         if np.all(target_subimage == source_subimage):
             print "SUCCESS: As we can see the values in the source pixel "
             print "         and the selected target pixels match as expected"
         else:
             print "ERROR: It looks like we have made some mistake."
```

```
Data loaded from source:
3

Data loaded from target:
[3 3 3 3]

SUCCESS: As we can see the values in the source pixel
         and the selected target pixels match as expected
```

Just to also show what we have done here visually, lets look at the maps of the elements we selected in source image—which shows that we selected the bottom-right pixel [1,1]—and the maps of the elements we selected in the target image—which shows that we selected the correspondign 4 pixel in the bottom-right corner of the target image.

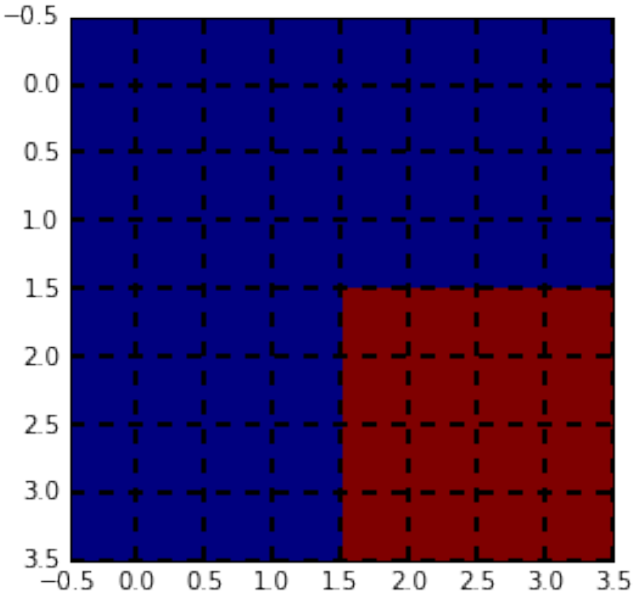
```
In [44]: print "Elements selected in the source image:"
         if HAS_PLT:
             source_selection_image = np.zeros(imr1_source.shape, dtype='bool')
             source_selection_image[source_selection] = True
             ax = plt.imshow(source_selection_image, interpolation='nearest')
             plt.grid(True, linestyle='--', linewidth=2)
```

```
Elements selected in the source image:
```




```
In [45]: print "Elements selected in the target image:"
         if HAS_PLT:
             target_selection_image = np.zeros(imr1_target.shape, dtype='bool')
             target_selection_image[target_selection[...,0,:], target_selection[...,1,:]] = True
             ax = plt.imshow(target_selection_image, interpolation='nearest')
             plt.grid(True, linestyle='--', linewidth=2)
```

```
Elements selected in the target image:
```



Now when we loaded the data from the target image we used the following notation:
`target_subimage = imr1_target[:][target_selection[...,0,:] , target_selection[...,1,:] ]`

This notation is not arbitrary but we can deterimine programmatically based on the specification of the relationship how we need to define this selection:

Why are we doing `imr1_target[:]`?: Again, we here load the full data of the target image first mainly because h5py.Dataset does not support the same fancy array-based indexing methods that numpy supports, i.e., this is just a quick workaround and we could also load the data one-element-at-a-time

Why are we doing `[target_selection[...,0,:] , target_selection[...,1,:] ]`: Our target dataset is two-dimensional—which we can check by looking at `imr1['MAP_TO_TARGET'].target.shape`. This means our indicies have multiple components and in numpy we need to describe the elements we select separately for each dimension of our data. Hence the two components: i) `target_selection[...,0,:]` and ii) `target_selection[...,1,:]`

Ok, but how do we know we need to slice `target_selection[...,0,:]` in this particular order? Here we need to look at the specification of the `axis` and `shape` of the relationship from our `map` to our `target`

```
In [46]: print imr1['MAP_TO_TARGET'].source_axis    # These are the axis used for indexing
         print imr1['MAP_TO_TARGET'].source.shape   # This is the shape of our map dataset

{u'INDEXING_AXIS': 2, u'STACK_AXIS': 3}
(2, 2, 2, 4)
```

as we can see: 1. `INDEXING_AXIS` is the second-to-last axis of our map dataset, i.e., this is the axis that stores the components of our mutlti-dimensional indicies. 2. `STACK_AXIS` is the last axis of our map dataset, i.e., the set of elements that we map to are stored last.

Hence, when we select the data in our target datsets, the first dimensions—which are describes by `...`, of our selection— are the intrinsic dimensions of our map dataset. The **0** then is the dimension in our `target` dataset. And, finally `:` indicates that we want to select all elements that we map to (from our `STACK_AXIS`). Therefore, we write `target_selection[...,0,:]`.

In the above example we selected just a single element `[1,1]` from our source data. As such the `target_selection[...,0,:]` addresses all data, because, `...`, collapes to nothing. However, if we select multiple elements from our source, then we will have to iterate over our intrinsic dimensions and load the mapped elements for each source element independently. An example for this is given below in the Summary section below.

Summary: Interacting with Index Map Relationships Since the above descriptions are heavly interlaced with validation code, below a quick summary of the main calls we used to interact with our index map relationship.

To create our index map relationship we used `RelationshipAttribute.create_index_map_relationship(...)` as follows:

```
mapping_relationships = RelationshipAttribute.create_index_map_relationship(
        name='upsampled_image_relationship',
        map_object=map_source_to_target,
        source_object=source_image,
        target_object=target_image,
        map_indexing_axis=2,
        map_stack_axis=3)
```

```
In [47]: # 1) Get all index map relationships
         imr_names =  RelationshipAttribute.get_index_map_relationship_names(
             parent_object=source_image)
```

```
In [48]: # 2) Get specifically the relationships that desribe the
         # 'upsampled_image_relationship' index map relationship
         imr1 = RelationshipAttribute.get_index_map_relationship(
             parent_object=source_image,
             relationship_name='upsampled_image_relationship')

In [49]: # 3) Get the source, target and map dataset
         imr1_source = imr1['MAP_TO_SOURCE'].target
         imr1_target = imr1['MAP_TO_TARGET'].target
         imr1_map = imr1['MAP_TO_SOURCE'].source

In [50]: # 4) Load data from the source
         # 4.1) Create the selection. We do this just for convenience but
         #      we could also just write [0:1,:] every time
         source_selection = np.s_[0:1, :]

         # 4.2) If the relationship applies to the whole source dataset---i.e.,
         #      'if imr1['SOURCE_TO_MAP'].source_axis is None' or alternatively
         #      'if imr1['MAP_TO_SOURCE'].target_axis is None---then we can
         #      simply slice agains the source directly.
         selected_source_data1 = imr1_source[source_selection]

         # 4.3) If the index map relationship applies only to a subset of the source
         #      dataset, then we need to look at imr1['SOURCE_TO_MAP'].source_axis
         #      to identify to which axis in the source we can select from

In [51]: # 5) Load data from the target
         # 5.1) Map the selection from the source to the target
         target_selection = imr1['MAP_TO_TARGET'][source_selection]

         # 5.2) For each element we selected in the source dataset, load
         #      the corresponding data values in the target dataset
         selected_target_data = []
         for i in range(target_selection.shape[0]):
             for j in range(target_selection.shape[1]):
                 pixel_data = imr1_target[:][target_selection[i,j,0,:] , target_selection[i,j,1,:]]
                 selected_target_data.append(pixel_data)
```

As we can see, our index map relationship allows us to programatically interact with both our source and target image dataset without having to know a priori: i) the name and location of our datasets (i.e., we need to know only the source or the map dataset), ii) the shapes of our datasets, iii) the ordering of elements in our map.

4.2 Other Relationship Chains

Currently `index map relationships` are the only kind of relationship chains with direct support by the BRAINformat API but users may, naturally, define their own semantics for other kinds of relationship chains (as long as they manage those chains).

This page intentionally left blank.

2 SUPPLEMENT: FORMAT SPECIFICATION, LICENSE, COPYRIGHT

1 Format Specifications

1.1 Specifying File Modules

Managed file modules are specified via Python dictionaries (which may be serialized to JSON). Below we describe the syntax of how to specify the various file modules in detail. To help with the creation of valid specifications, we also provide a series of classes that help with the incremental creation of new file module specification. Here a quick overview of the main specification modules and classes:

File Format Specification Helper Classes

`brain.dataformat.spec.BaseSpec`()	
`brain.dataformat.spec.DatasetSpec`(dataset, ...)	Specification of a dataset
`brain.dataformat.spec.GroupSpec`(group, ...)	Specification of a group
`brain.dataformat.spec.FileSpec`(description)	Specification of a file
`brain.dataformat.spec.ManagedSpec`(format_type)	Specification of a contained Managed Object
`brain.dataformat.spec.AttributeSpec`(...[, ...])	Specification of an attribute
`brain.dataformat.spec.DimensionSpec`(name, ...)	Specification of a dimension scale
`brain.dataformat.spec.RelationshipSpec`(...)	Specification of a relationship between datasets
`brain.dataformat.spec.RelationshipTargetSpec`(...)	Specification of the target of a relationship

Group Specification

The group specification must contain the following keys/values:

- **datasets** : Dictionary of dataset specifications (see below) describing the datasets contained in the group. Note, these are datasets that are managed directly by this managed group. Datasets with a dedicated manager type as part of the file format are specified via the **managed_objects** key.
- **groups** : Dictionary of group specification describing the groups contained in this group. Note, these are groups that are managed directly by this managed group. Groups with a dedicated manager type as part of the file format API are specified via the **managed_objects** key.
- **managed_objects** : List of managed object specification describing additional managed datasets or groups contained in this group.
- **attributes** : List of attribute specifications. These are attributes associated with the group object directly.
- **group** : The name of the group. May be None in case the group does not have a fixed name but multiple instances of the managed group object may exist, in which case the **prefix** key should be set. In general, only one of **group** or **prefix** should be set but not both.
- **prefix** : String indicating the prefix to be used for the group name. The prefix is used in case that multiple instances of this managed object are allowed.
- **optional** : Boolean indicating whether the group is optional or mandatory.
- **description** : String describing the purpose of this managed object type. Stored in the brainformat_description attribute used to help new users with the interpretation of the format.
- **relationships** : Optional list of relationship specifications describing relationships of this group to other objects. The key may be omitted if no relationships are specified.

Example:

```
{'datasets': {'ecog_data': {'dataset': 'raw_data', # dataset key is mandatory may be None
                            'prefix': None,        # prefix key is mandatory may be None
                            'optional': False,     # optional key is mandatory
                            # dimensions key is optional. If specified we assume that the number of
                            # dimensions is fixed and that a scale is defined for all dimensions,
                            # even if it is empty
                            # NOTE: if multiple scales are defined for the same axis, then the name
                            #       for those axis must be the same, while the unit key may differ
                            #       between scales for the same dimension
                            'dimensions': [{'name': 'space',     # Mandatory
                                            'unit': 'id',        # Mandatory
                                            'optional': False,   # Mandatory
                                            'dataset': 'electrode_id',  # Mandatory. Set to None tc
                                            'axis': 0,
                                            'description': 'description': 'Id of the recording elec
                                           {'name': 'time',
                                            'unit': 'ns',
                                            'optional': False,
                                            'dataset': 'time_axis',
                                            'axis': 1,
                                            'description': 'Sample time in ns'}],  # use empty dict
                            'description': 'Dataset with the ECoG recordings data',  # Mandatory
                            'attributes': [{'attribute': 'unit',   # Mandatory but may be empty
                                            'value': 'Volt', # Mandatory may be None to indicate us
                                            'prefix': None,        # Mandatory
                                            'opional': False}]},   # Mandatory
              'sampling_rate': {'dataset': 'sampling_rate',
                                'prefix': None,
                                'optional': False,
                                'attributes': [{'attribute': 'unit',
                                                'value': 'KHz',
                                                'prefix': None,
                                                'optional': False}],
                                'description': 'Sampling rate in KHz'},
              'layout': {'dataset': 'layout',
                         'prefix': None,
                         'optional': True,
                         'dimensions': [],
                         'attributes': [],
                         'description': 'The physical layout of the electrodes.'}},
 'groups': {},           # Mandatory
 'managed_objects': [],  # e.g., {'format_type': 'BrainDataECoG', 'optional': True}
 'attributes': [],  # e.g, [{'attribute': 'unit', 'value': KHz, 'prefix': None, 'optional': False}]
 'group': None,          # Mandatory (use 'dataset' in case of a managed dataset
 'prefix': "ecog_data_",
 'optional': False,
 'description': 'Managed group for storage of raw ECoG recordings.'}
```

File Specification

The specification of managed files is very similar to the specification of group objects with the following key differences:

- **file_prefix** : Required additional entry describing the name prefix for filenames. May be set to *None* indicating that arbitrary filenames may be used.
- **file_extension** : Required file extension. May be set to *None* to indicate that arbitrary file may be used exten-

sions.

- **group**, **prefix** : The behavior of these keys is identical to groups only that they are used to determine the name of external links to the files root group. In contrast to group specification, both *group* and *prefix* are allowed to be simultaneously set to None, indicating that no external links should be generated to the file. This is, e.g,the for *brain.dataformat.base.ManagedObjectFile* which is a pure container object with the intend that we should only link to specific object within the container but not the container file itself.

```
{'datasets': {},
 'groups': {},
 'managed_objects': [{'format_type': 'BrainDataData', 'optional': False},
                     {'format_type': 'BrainDataDescriptors', 'optional': False}],
 'attributes': [],
 'group': None,
 'prefix': "entry_",
 'file_prefix': None,
 'file_extension': '.h5',
 'optional': False,
 'description': 'Managed BRAIN file.'}
```

Managed Objects Specification

The specification of managed objects consists of the following keys/values:

- **format_type** : String indicating the type of the managed object, e.g., **BrainDataECoG**. Use **ManagedObject** as format type, to indicate that any type of managed object may be part of the current group or file.
- **optional**: Boolean indicating whether the managed object optional or mandatory. This overwrites the **optional** key of the format specification of the specification of the managed object.

Example managed object specification:

```
{'format_type': 'BrainDataECoG', 'optional': True}
```

Attribute Specification

The specification of attributes consists of the following keys/values:

- **attribute** : Fixed name for the attribute. May be None in case that a **prefix** is specified allowing multiple instances of the attribute for the same object.
- **value** : Value of the attribute. May be None in case the value for the attribute is not fixed by user-defined.
- **prefix** : Prefix for the attribute. The prefix is automatically appended by a number so that multiple instances of the attribute are possible.
- **optional**: Boolean indicating whether the managed object is optional or mandatory.
- **description**: Optional string describing the attribute in a human-readable form. The description is optional for attributes, as it cannot be saved to file as part of the attribute itself (attributes cannot have additional attributes) but only as part of the larger spec of the object the attribute is applied to

Example attribute specification:

```
{'attribute': 'unit',     # Mandatory but may be None is 'prefix' is set
 'value': 'Volt',         # Mandatory may be None to indicate that the value user defined (rather th
 'prefix': None,          # Mandatory may be None if 'attribute' is set
 'optional': False}    # Mandatory boolean.
```

Dataset Specification

The specification of datasets consists of the following keys/values:

- **dataset** : Fixed name for the dataset. May be None in case that **prefix** is specified to indicate that multiple numbered instances of the dataset type are allowed.
- **prefix** : String indicating the prefix to be used for the dataset name. The prefix is used in case that multiple instances of this dataset object are allowed.
- **dimensions** : List of dimension scale specification. The **dimensions** key is optional. If specified we assume that the number of dimensions is fixed and that a scale is defined for all dimensions, even if it is empty. NOTE: if multiple scales are defined for the same axis, then the name for those axis must be the same, while the unit key may differ between scales for the same dimension
- **dimensions_fixed** Boolean indicating whether the dataset must have exactly the number of dimensions specified by dimensions. If False, then the dataset is allowed to have additional dimensions not specified in *dimensions*. This parameter is optional. If the parameter is missing and *dimensions* are specified then it is implicitly assumed to be set to True. If the parameter is missing and *dimensions* is empty then the parameter is assumed to be implicitly False.
- **description** : String describing the purpose of this managed object type. Stored in the brainformat_description attribute used to help new users with the interpretation of the format.
- **attributes** : List of attribute specifications. These are attributes associated with the group object directly.
- **optional**: Boolean indicating whether the object is optional or mandatory.
- **primary** : Boolean indicating whether the dataset is a primary data source for analysis. This attribute is optional and is assumed to be False if missing. Marking primary data sources is useful when using data files as part of a third-party visualization and analysis tools and allows third-party tools to discover which datasets are the primary sources.
- **relationships** : Optional list of relationship specifications describing relationships of this dataset to other objects. The key may be omitted if no relationships are specified.

Example dataset specification:

```
{'dataset': 'raw_data',
 'prefix': None,
 'optional': False,
 'primary': True
 'dimensions': [{'name': 'space',
                 'unit': 'id',
                 'optional': False,
                 'dataset': 'electrode_id',
                 'axis': 0},
                {'name': 'time',
                 'unit': 'ns',
                 'optional': False,
                 'dataset': 'time_axis',
                 'axis': 1}],
 'description': 'Dataset with the ECoG recordings data',
 'attributes': [{'attribute': 'unit',
                 'value': 'Volt',
                 'prefix': None,
                 'opional': False}]}
```

Dimension Scales Specification

Dimension scale specifications are an optional part of dataset specifications and are only allowed there. Dimension scales describe the name/type of a particular dimension of dataset. The specification of dimension scales consist of the following keys/values:

- **name** : The name of the dimensions scale. NOTE: if multiple dimensions scales are associated with the same axis of a dataset, then their **name** must be identical while their **unit** keys may differ. If a dimension is required but does not have a dimension scale, then set the name to None. In this case unit and dataset should be None as well.
- **unit** : The units in which the dimension is expressed. May be None in case that only a name for the dimensions should be specified but no actual dimension scale. NOTE: if a dataset is specified then unit must be set as well as this is used to address the dataset.
- **optional**: Boolean indicating whether the object is optional or mandatory.
- **dataset** : The HDF5 dataset with the values for the dimension scale. May be None, in case that no actual axis scale should be specified, but rather only the dimensions should be labeled, but no actual dimensions scale should be created. In advanced cases, this may also be a complete Dataset Specification (rather than just a name).
- **axis** : Unsigned Integer indicating the axis/dimension the dimension scale is associated with. (Mandatory)
- **description**: Description of the dimensions scale. The description is associated with the dataset as the format description attribute (i.e., if the dataset is not set to None). (Mandatory)
- **relationships** : Optional list of relationship specifications describing relationships of the dataset associated with the dimensions scale to other objects. The key may be omitted if no relationships are specified. If relationships are specified, then the **dataset** key must be set.

```
{'name': 'space',
 'unit': 'id',
 'optional': False,
 'dataset': 'electrode_id',
 'axis': 0,
 'description': 'Id of the recording electrode'}
```

Relationship Specification

Relationships describe semantic links between file objects (usually datasets or groups). They are an optional part of Dataset and Group specifications. Relationship may also be defined as part of Dimensions Scale specifications if a dataset is associated with the scale.

In practice, many relationships are dynamic (i.e., only known once the data is generated), however, some relationships can already be described in the specification of the file format itself. Such static relationships often describe basic details of data structures, e.g, one array storing indices into another array etc.. The specification of basic relationships consists of the following keys/values:

- **attribute**: The name of the attribute used to store the relationship. May be None if *prefix* is specified. One of *prefix* or *attribute* must be set.
- **prefix**: Prefix of attributes used to store this type of relationship. May be None if *attribute* is specified. One of *prefix* or *attribute* must be set.
- **traget**: Dictionary specifying the target object of the relationship (e.g, the dataset the relationship points to). The target dictionary consists of the following keys/values:
  - **filename** File where the target is located. Set to None in case that the target is located in the same file as the source.

  - **global_path** Typically we try to define Managed Objects in a self-contained fashion, i.e., all data related to object should be available in the same managed objects. However, in some cases it is useful to store repeatably-used and shared data in central locations. The global key allows us to specify the location of global targets that we want to point to. The *dataset*, *group*, *prefix* keys are still honored, i.e, the global key only gives us the base location where the object is located.
    * <*path*> The global key may be an absolute path, in which case the path will start with */*. This form of description is often used when a user adds custom relationships to the file.
    * <*GlobalTargetClass*>*:*<*GlobalKey*> *GlobalTargetClass* indicates the name of the dictionary (i.e, namespace) for the global targets and the *GlobalKey* is the name of the particular global target, where the value associated with the key is expected to be the absolute path within the file to the global target. This strategy is mainly useful when specifying static relationships and assumes that a dictionary of GLOBAL_PATH is available for the format so that these paths can be resolved to absolute paths when creating the relationships. I.e., within a file, *global_paths* should always be an absolute <*path*>, but within a specification of a format this strategy may be used to avoid the inclusing of absolute path in the specification of indiviual Managed Objects and to allow global paths to be specified in a central location that the API must be aware of.
    * *None* indicating that we are indexing a local structure within the parent group of the source object
  - **dataset** The name of the dataset the relationship points to. This is usually a relative name within the current specification (i.e., the parent object that contains the object with the relationship). May be None if *group* or *prefix* are defined. *dataset* must be None if *group* or *prefix* are set.
  - **group** The name of the group the relationship points to. This is usually a relative name within the current specification (i.e., the parent object that contains the object with the relationship). May be None if *dataset* or *prefix* are defined. Must be None if *dataset* is set. May be used in combindation with *prefix* to point to the parent location where the prefix's are located.
  - **prefix** The name prefix of the dataset or group the relationship points to . This is usually a relative name within the current specification (i.e., the parent object that contains the object with the relationship). May be None if *group* or *dataset* are defined. May be used in combination with *group*.
  - **axis** The index (or name) of the axis we point to (if the relationship points to a particular dataset) or None. Must be None if the relationship points to a group. May be a list of axis indicies if the relationship encompasses multiple axis.

- **axis**: The axis of the source object the relationships refers to (if the source is a dataset). May be a list of axis indicies if the relationship encompasses multiple axis. Use None if the relationship does not refer to a particular axis but rather the source object as a whole. (Must be None if the relationship refers to a group). In the case of *indexes* relationships, axis may be used to identify the dimension that defines the indicies (e.g, if we index a 2D dataset then we may have a 20x2 dataset containing 20 two-dimensional indicies, where the second axis defines the indicies). Also, in *indexes* relationships, this may also be a dict to encode the {'INDEXING_AXIS':<value>, 'STACK_AXIS':<value>}, i.e., the axis used for indexing and the axis used for stacking to describe 1 to many indexing.
- **relationship_type**: String indicating the type of the relationship. Currently supported relationship types include:
  - *indexes*: The *source* dataset contains indices into the target dataset. These are often integer indices, however, e.g, if the relationship points to a group, then the source dataset may also contain strings selecting the objects stored in the group. The source of such a relationship, however, should always be a dataset. In the case of multi-dimensions indicies it is useful to specify *axis*

for the *source* to indicate along which dimensions (typically 0 or the last dimensions) indices are stored.

- *indexes_values*: The *source* selects certain parts of the *target* based on the values (or keys in case of group(s)) in the *target*. Specification of axis for the *target* usually does not make sense for this type of relationship. The *indexes_values* relationship implies that the datasets use a *shared_encoding* (see next bullet) and is effectively a special type of *shared_encoding* relationship that beyond the encoding describes that the *source* is selecting data in the *target* based on value.
- *shared_encoding*: The target and source dataset contain values with the same encoding, i.e., values in the two datasets can be directly compared. The specification of a target axis usually does not make sense for this type of relationship.
- *shared_ascending_encoding* : Same as *shared_encoding* but the source and target datasets are expected to be sorted by value (e.g., in the case of time)
- *order*: The ordering of objects matches between the datasets along the given axes. This relationship type in practice makes mainly sense between datasets as no explicit order is defined for objects inside a group (however a user may impose a particular order in their formats if desired).
- *equivalent*: In addition to order, this relationship type expresses that the source and target object encode the same data (even if they might store different values). E.g., a token may be encoded by its name or by an integer index. This relationship also implies that the source and target contain the same number of values ordered in the same fashion.
- *user*: Arbitrary user-defined relationship. Use this type to describe arbitrary relationships between objects. E.g, two datasets may have some semantic relationship that a user may want to document. Additional data to describe this relationship may be stored in the *properties* of the relationship.

- **optional**: Boolean indicating whether the object is optional or mandatory
- **description**: Text describing the relationship in a human-readable form
- **properties**: Optional (JSON serializable) dictionary with additional user properties.

NOTE: In the case of relationships we assume that both the *source* and *target* have already been specified, i.e., we can here only refer to the data by name and not via Dataset or Group specifications.

NOTE: Relationships may generally only be defined for Datasets and Groups (including Datasets of DimensionScales) but not for Attributes.

NOTE: When storing Relationship Attributes the BRAINformat API prepends a fixed prefixed defined in *RelationshipAttribute.RELATIONSHIP_ATTRIBUTE_PREFIX* to the attribute name. This is to standardize and ease finding relationship attributes.

NOTE: The BRAINformat API also implements *index map relationships* which are a series of relationships used in conjunction to describe the mapping between two datasets via an intermediary map dataset. Relationship attributes that belong to such a relationship are identified via a set of postfixes defined in *RelationshipAttribute.INDEX_MAP_RELATIONSHIP_POSTFIX*. Further details about index map relationships can be found in the introductory tutorial on relationship attributes available as part of the *brain.examples* tutorial series.

```
{'attribute':'region_encoding',          # The name of the relationship
 'prefix': None,                         # Attribute is set
 'target': {'global': None,              # None if this is a local relationship or <GlobalTargetCla
            'dataset': 'anatomy_id',     # The dataset we link to
            'group': None,               # The group we link to
            'prefix': None,              # The prefix of the object(s) we link to
            'prefix_index': None,        # The index of the prefix object we link to or None if we
            'axis': None},               # The axis of the dataset we link to
```

```
 'axis': None,                          # The axis that has the relationship
 'type': 'equivalent',                  # The type of the relationship.
 'optional': False,                     # Is the relationship optional
 'description':' Region encoded as id', # Text description of the relationship
 'properties': None                     # Optional dict with user properties
}
```

2 Full Specification for BrainDataFile (JSON)

Since the format is directly specified by the ManagedObjects we can easily construct the full specification for the full file format (e.g, for *BrainDataFile*) as follows:

```
import time
from brain.dataformat.brainformat import BrainDataFile
from brain.dataformat.spec import *

format_spec = BrainDataFile.get_format_specification_recursive()
file_spec = BaseSpec.from_dict(format_spec)

print '**' + str(time.ctime(time.time())) + '**'
print file_spec.to_json(pretty=True)
```

An example output from the above code-example is given below. Similarly we can compute the specification of any managed sub-object of a file.

```
**Tue Apr 24 23:54:59 2015**
{
    "attributes": [
        {
            "attribute": "format_type",
            "optional": false,
            "prefix": null,
            "value": "BrainDataFile"
        },
        {
            "attribute": "format_description",
            "optional": false,
            "prefix": null,
            "value": "Managed BRAIN file."
        },
        {
            "attribute": "object_id",
            "optional": true,
            "prefix": null,
            "value": null
        },
        {
            "attribute": "format_specification",
            "optional": false,
            "prefix": null,
            "value": null
        }
    ],
    "datasets": {},
    "description": "Managed BRAIN file.",
    "file_extension": ".h5",
```

```
"file_prefix": null,
"group": null,
"groups": {
    "data": {
        "attributes": [
            {
                "attribute": "format_type",
                "optional": false,
                "prefix": null,
                "value": "BrainDataData"
            },
            {
                "attribute": "format_description",
                "optional": false,
                "prefix": null,
                "value": "Managed group for storage of brain data (internal and external)."
            },
            {
                "attribute": "object_id",
                "optional": true,
                "prefix": null,
                "value": null
            },
            {
                "attribute": "format_specification",
                "optional": false,
                "prefix": null,
                "value": null
            }
        ],
        "datasets": {},
        "description": "Managed group for storage of brain data (internal and external).",
        "group": "data",
        "groups": {
            "external": {
                "attributes": [
                    {
                        "attribute": "format_type",
                        "optional": false,
                        "prefix": null,
                        "value": "BrainDataExternalData"
                    },
                    {
                        "attribute": "format_description",
                        "optional": false,
                        "prefix": null,
                        "value": "Managed group for storage of external data related to the int
                    },
                    {
                        "attribute": "object_id",
                        "optional": true,
                        "prefix": null,
                        "value": null
                    },
                    {
                        "attribute": "format_specification",
                        "optional": false,
                        "prefix": null,
```

```
                "value": null
            }
        ],
        "datasets": {},
        "description": "Managed group for storage of external data related to the inter
        "group": "external",
        "groups": {},
        "managed_objects": [],
        "optional": false,
        "prefix": null,
        "relationships": []
    },
    "internal": {
        "attributes": [
            {
                "attribute": "format_type",
                "optional": false,
                "prefix": null,
                "value": "BrainDataInternalData"
            },
            {
                "attribute": "format_description",
                "optional": false,
                "prefix": null,
                "value": "Managed group for storage of a collection of internal brain d
            },
            {
                "attribute": "object_id",
                "optional": true,
                "prefix": null,
                "value": null
            },
            {
                "attribute": "format_specification",
                "optional": false,
                "prefix": null,
                "value": null
            }
        ],
        "datasets": {},
        "description": "Managed group for storage of a collection of internal brain dat
        "group": "internal",
        "groups": {
            "ecog_data_": {
                "attributes": [
                    {
                        "attribute": "format_type",
                        "optional": false,
                        "prefix": null,
                        "value": "BrainDataECoG"
                    },
                    {
                        "attribute": "format_description",
                        "optional": false,
                        "prefix": null,
                        "value": "Managed group for storage of raw ECoG recordings."
                    },
                    {
```

```
            "attribute": "object_id",
            "optional": true,
            "prefix": null,
            "value": null
        },
        {
            "attribute": "format_specification",
            "optional": false,
            "prefix": null,
            "value": null
        }
    ],
    "datasets": {
        "ecog_data": {
            "attributes": [
                {
                    "attribute": "unit",
                    "optional": false,
                    "prefix": null,
                    "value": "Volt"
                }
            ],
            "dataset": "raw_data",
            "description": "Dataset with the ECoG recordings data",
            "dimensions": [
                {
                    "axis": 0,
                    "dataset": "electrode_id",
                    "description": "Id of the recording electrode",
                    "name": "space",
                    "optional": false,
                    "relationships": [],
                    "unit": "id"
                },
                {
                    "axis": 1,
                    "dataset": "time_axis",
                    "description": "Sample time in ms",
                    "name": "time",
                    "optional": false,
                    "relationships": [],
                    "unit": "ms"
                },
                {
                    "axis": 0,
                    "dataset": "anatomy_name",
                    "description": "Name of region location of the electrod
                    "name": "space",
                    "optional": true,
                    "relationships": [],
                    "unit": "region name"
                },
                {
                    "axis": 0,
                    "dataset": "anatomy_id",
                    "description": "Integer id of the region location of th
                    "name": "space",
                    "optional": true,
```

```
                    "relationships": [],
                    "unit": "region id"
                }
            ],
            "dimensions_fixed": true,
            "optional": false,
            "prefix": null,
            "primary": true,
            "relationships": []
        },
        "layout": {
            "attributes": [],
            "dataset": "layout",
            "description": "The physical layout of the electrodes.",
            "dimensions": [],
            "optional": true,
            "prefix": null,
            "relationships": []
        },
        "sampling_rate": {
            "attributes": [
                {
                    "attribute": "unit",
                    "optional": false,
                    "prefix": null,
                    "value": "Hz"
                }
            ],
            "dataset": "sampling_rate",
            "description": "Sampling rate in Hz",
            "dimensions": [],
            "optional": false,
            "prefix": null,
            "relationships": []
        }
    },
    "description": "Managed group for storage of raw ECoG recordings.",
    "group": null,
    "groups": {
        "annotations_": {
            "attributes": [
                {
                    "attribute": "collection_description",
                    "optional": false,
                    "prefix": null,
                    "value": null
                },
                {
                    "attribute": "format_type",
                    "optional": false,
                    "prefix": null,
                    "value": "AnnotationDataGroup"
                },
                {
                    "attribute": "format_description",
                    "optional": false,
                    "prefix": null,
                    "value": "Managed group for storage of a collection of
```

```
        },
        {
            "attribute": "object_id",
            "optional": true,
            "prefix": null,
            "value": null
        },
        {
            "attribute": "format_specification",
            "optional": false,
            "prefix": null,
            "value": null
        }
    ],
    "datasets": {
        "annotation_type_indexes": {
            "attributes": [],
            "dataset": "annotation_type_indexes",
            "description": "Dataset indicating for each selection t
            "dimensions": [
                {
                    "axis": 0,
                    "dataset": null,
                    "description": "Integer index into the annotati
                    "name": "type_index",
                    "optional": false,
                    "relationships": [],
                    "unit": null
                }
            ],
            "dimensions_fixed": true,
            "optional": false,
            "prefix": null,
            "relationships": []
        },
        "annotation_types": {
            "attributes": [],
            "dataset": "annotation_types",
            "description": "List of all available annotation types"
            "dimensions": [
                {
                    "axis": 0,
                    "dataset": null,
                    "description": "Integer index of the type",
                    "name": "type_index",
                    "optional": false,
                    "relationships": [],
                    "unit": null
                }
            ],
            "dimensions_fixed": true,
            "optional": false,
            "prefix": null,
            "relationships": []
        },
        "data_object": {
            "attributes": [],
            "dataset": "data_object",
```

```
        "description": "None",
        "dimensions": [],
        "optional": false,
        "prefix": null,
        "relationships": []
    },
    "descriptions": {
        "attributes": [],
        "dataset": "descriptions",
        "description": "Dataset with the annotation description
        "dimensions": [
            {
                "axis": 0,
                "dataset": null,
                "description": "Integer index of the annotation
                "name": "annotation_index",
                "optional": false,
                "relationships": [],
                "unit": null
            }
        ],
        "dimensions_fixed": true,
        "optional": false,
        "prefix": null,
        "relationships": []
    },
    "properties": {
        "attributes": [
            {
                "attribute": "name",
                "optional": false,
                "prefix": null,
                "value": null
            }
        ],
        "dataset": null,
        "description": "Datasets with a particular property for
        "dimensions": [
            {
                "axis": 0,
                "dataset": null,
                "description": "Integer index of the selection"
                "name": "selection_index",
                "optional": false,
                "relationships": [],
                "unit": null
            }
        ],
        "dimensions_fixed": false,
        "optional": true,
        "prefix": "property_",
        "relationships": []
    },
    "selection_indexes": {
        "attributes": [],
        "dataset": "selection_indexes",
        "description": "Dataset indicating for each axis the in
        "dimensions": [
```

```
                {
                    "axis": 0,
                    "dataset": null,
                    "description": "Integer index of the annotation
                    "name": "annotation_index",
                    "optional": false,
                    "relationships": [],
                    "unit": null
                },
                {
                    "axis": 1,
                    "dataset": "axis_index",
                    "description": "Integer index of the axis",
                    "name": "axis_index",
                    "optional": false,
                    "relationships": [],
                    "unit": "index"
                }
            ],
            "dimensions_fixed": true,
            "optional": false,
            "prefix": null,
            "relationships": []
        },
        "selections": {
            "attributes": [
                {
                    "attribute": "axis",
                    "optional": false,
                    "prefix": null,
                    "value": null
                }
            ],
            "dataset": null,
            "description": "Datasets with all selections for the in
            "dimensions": [
                {
                    "axis": 0,
                    "dataset": null,
                    "description": "Integer index of the selection"
                    "name": "selection_index",
                    "optional": false,
                    "relationships": [],
                    "unit": null
                }
            ],
            "dimensions_fixed": false,
            "optional": false,
            "prefix": "selections_axis_",
            "relationships": []
        }
    },
    "description": "Managed group for storage of a collection of an
    "group": null,
    "groups": {},
    "managed_objects": [],
    "optional": true,
    "prefix": "annotations_",
```

```
                "relationships": []
            }
        },
        "managed_objects": [],
        "optional": true,
        "prefix": "ecog_data_",
        "relationships": []
    },
    "ecog_data_processed_": {
        "attributes": [
            {
                "attribute": "format_type",
                "optional": false,
                "prefix": null,
                "value": "BrainDataECoGProcessed"
            },
            {
                "attribute": "format_description",
                "optional": false,
                "prefix": null,
                "value": "Managed group for storage of processed ECoG recording
            },
            {
                "attribute": "object_id",
                "optional": true,
                "prefix": null,
                "value": null
            },
            {
                "attribute": "format_specification",
                "optional": false,
                "prefix": null,
                "value": null
            }
        ],
        "datasets": {
            "ecog_data": {
                "attributes": [
                    {
                        "attribute": "unit",
                        "optional": false,
                        "prefix": null,
                        "value": null
                    },
                    {
                        "attribute": "original_name",
                        "optional": true,
                        "prefix": null,
                        "value": null
                    }
                ],
                "dataset": "processed_data",
                "description": "Dataset with the ECoG recordings data",
                "dimensions": [
                    {
                        "axis": 0,
                        "dataset": "spatial_id",
                        "description": "Id of the recording electrode",
```

```
        "name": "space",
        "optional": false,
        "relationships": [],
        "unit": "id"
    },
    {
        "axis": 1,
        "dataset": "time_axis",
        "description": "Sample time in ms",
        "name": "time",
        "optional": false,
        "relationships": [],
        "unit": "ms"
    },
    {
        "axis": 0,
        "dataset": "anatomy_name",
        "description": "Name of region location of the electrod
        "name": "space",
        "optional": true,
        "relationships": [],
        "unit": "region name"
    },
    {
        "axis": 0,
        "dataset": "anatomy_id",
        "description": "Integer id of the region location of th
        "name": "space",
        "optional": true,
        "relationships": [],
        "unit": "region id"
    },
    {
        "axis": 2,
        "dataset": "frequency_bands",
        "description": "Frequency bands of the channels",
        "name": "channels",
        "optional": true,
        "relationships": [],
        "unit": "Hz"
    },
    {
        "axis": 2,
        "dataset": "token_id",
        "description": "Integer Id of the token type",
        "name": "channels",
        "optional": true,
        "relationships": [],
        "unit": "token id"
    },
    {
        "axis": 2,
        "dataset": "token_name",
        "description": "Name of the token type",
        "name": "channels",
        "optional": true,
        "relationships": [],
        "unit": "token name"
```

```
                }
            ],
            "dimensions_fixed": true,
            "optional": false,
            "prefix": null,
            "primary": true,
            "relationships": []
        },
        "layout": {
            "attributes": [],
            "dataset": "layout",
            "description": "The physical layout of the electrodes.",
            "dimensions": [],
            "optional": true,
            "prefix": null,
            "relationships": []
        },
        "sampling_rate": {
            "attributes": [
                {
                    "attribute": "unit",
                    "optional": false,
                    "prefix": null,
                    "value": "Hz"
                }
            ],
            "dataset": "sampling_rate",
            "description": "Sampling rate in Hz",
            "dimensions": [],
            "optional": false,
            "prefix": null,
            "relationships": []
        }
    },
    "description": "Managed group for storage of processed ECoG recordings.
    "group": null,
    "groups": {
        "annotations_": {
            "attributes": [
                {
                    "attribute": "collection_description",
                    "optional": false,
                    "prefix": null,
                    "value": null
                },
                {
                    "attribute": "format_type",
                    "optional": false,
                    "prefix": null,
                    "value": "AnnotationDataGroup"
                },
                {
                    "attribute": "format_description",
                    "optional": false,
                    "prefix": null,
                    "value": "Managed group for storage of a collection of
                },
                {
```

```
            "attribute": "object_id",
            "optional": true,
            "prefix": null,
            "value": null
        },
        {
            "attribute": "format_specification",
            "optional": false,
            "prefix": null,
            "value": null
        }
    ],
    "datasets": {
        "annotation_type_indexes": {
            "attributes": [],
            "dataset": "annotation_type_indexes",
            "description": "Dataset indicating for each selection t
            "dimensions": [
                {
                    "axis": 0,
                    "dataset": null,
                    "description": "Integer index into the annotati
                    "name": "type_index",
                    "optional": false,
                    "relationships": [],
                    "unit": null
                }
            ],
            "dimensions_fixed": true,
            "optional": false,
            "prefix": null,
            "relationships": []
        },
        "annotation_types": {
            "attributes": [],
            "dataset": "annotation_types",
            "description": "List of all available annotation types"
            "dimensions": [
                {
                    "axis": 0,
                    "dataset": null,
                    "description": "Integer index of the type",
                    "name": "type_index",
                    "optional": false,
                    "relationships": [],
                    "unit": null
                }
            ],
            "dimensions_fixed": true,
            "optional": false,
            "prefix": null,
            "relationships": []
        },
        "data_object": {
            "attributes": [],
            "dataset": "data_object",
            "description": "None",
            "dimensions": [],
```

```
        "optional": false,
        "prefix": null,
        "relationships": []
    },
    "descriptions": {
        "attributes": [],
        "dataset": "descriptions",
        "description": "Dataset with the annotation description
        "dimensions": [
            {
                "axis": 0,
                "dataset": null,
                "description": "Integer index of the annotation
                "name": "annotation_index",
                "optional": false,
                "relationships": [],
                "unit": null
            }
        ],
        "dimensions_fixed": true,
        "optional": false,
        "prefix": null,
        "relationships": []
    },
    "properties": {
        "attributes": [
            {
                "attribute": "name",
                "optional": false,
                "prefix": null,
                "value": null
            }
        ],
        "dataset": null,
        "description": "Datasets with a particular property for
        "dimensions": [
            {
                "axis": 0,
                "dataset": null,
                "description": "Integer index of the selection"
                "name": "selection_index",
                "optional": false,
                "relationships": [],
                "unit": null
            }
        ],
        "dimensions_fixed": false,
        "optional": true,
        "prefix": "property_",
        "relationships": []
    },
    "selection_indexes": {
        "attributes": [],
        "dataset": "selection_indexes",
        "description": "Dataset indicating for each axis the in
        "dimensions": [
            {
                "axis": 0,
```

```
                        "dataset": null,
                        "description": "Integer index of the annotation
                        "name": "annotation_index",
                        "optional": false,
                        "relationships": [],
                        "unit": null
                    },
                    {
                        "axis": 1,
                        "dataset": "axis_index",
                        "description": "Integer index of the axis",
                        "name": "axis_index",
                        "optional": false,
                        "relationships": [],
                        "unit": "index"
                    }
                ],
                "dimensions_fixed": true,
                "optional": false,
                "prefix": null,
                "relationships": []
            },
            "selections": {
                "attributes": [
                    {
                        "attribute": "axis",
                        "optional": false,
                        "prefix": null,
                        "value": null
                    }
                ],
                "dataset": null,
                "description": "Datasets with all selections for the in
                "dimensions": [
                    {
                        "axis": 0,
                        "dataset": null,
                        "description": "Integer index of the selection"
                        "name": "selection_index",
                        "optional": false,
                        "relationships": [],
                        "unit": null
                    }
                ],
                "dimensions_fixed": false,
                "optional": false,
                "prefix": "selections_axis_",
                "relationships": []
            }
        },
        "description": "Managed group for storage of a collection of an
        "group": null,
        "groups": {},
        "managed_objects": [],
        "optional": true,
        "prefix": "annotations_",
        "relationships": []
    }
```

```
                    },
                    "managed_objects": [],
                    "optional": true,
                    "prefix": "ecog_data_processed_",
                    "relationships": []
                }
            },
            "managed_objects": [],
            "optional": false,
            "prefix": null,
            "relationships": []
        }
    },
    "managed_objects": [],
    "optional": false,
    "prefix": null,
    "relationships": []
},
"descriptors": {
    "attributes": [
        {
            "attribute": "format_type",
            "optional": false,
            "prefix": null,
            "value": "BrainDataDescriptors"
        },
        {
            "attribute": "format_description",
            "optional": false,
            "prefix": null,
            "value": "Managed group for storage of a collection of brain data descriptors."
        },
        {
            "attribute": "object_id",
            "optional": true,
            "prefix": null,
            "value": null
        },
        {
            "attribute": "format_specification",
            "optional": false,
            "prefix": null,
            "value": null
        }
    ],
    "datasets": {},
    "description": "Managed group for storage of a collection of brain data descriptors.",
    "group": "descriptors",
    "groups": {
        "dynamic": {
            "attributes": [
                {
                    "attribute": "format_type",
                    "optional": false,
                    "prefix": null,
                    "value": "BrainDataDynamicDescriptors"
                },
                {
```

```
                "attribute": "format_description",
                "optional": false,
                "prefix": null,
                "value": "Managed group for storage of static descriptors."
            },
            {
                "attribute": "object_id",
                "optional": true,
                "prefix": null,
                "value": null
            },
            {
                "attribute": "format_specification",
                "optional": false,
                "prefix": null,
                "value": null
            }
        ],
        "datasets": {},
        "description": "Managed group for storage of static descriptors.",
        "group": "dynamic",
        "groups": {},
        "managed_objects": [],
        "optional": false,
        "prefix": null,
        "relationships": []
    },
    "static": {
        "attributes": [
            {
                "attribute": "format_type",
                "optional": false,
                "prefix": null,
                "value": "BrainDataStaticDescriptors"
            },
            {
                "attribute": "format_description",
                "optional": false,
                "prefix": null,
                "value": "Managed group for storage of static descriptors."
            },
            {
                "attribute": "object_id",
                "optional": true,
                "prefix": null,
                "value": null
            },
            {
                "attribute": "format_specification",
                "optional": false,
                "prefix": null,
                "value": null
            }
        ],
        "datasets": {},
        "description": "Managed group for storage of static descriptors.",
        "group": "static",
        "groups": {},
```

```
                    "managed_objects": [],
                    "optional": false,
                    "prefix": null,
                    "relationships": []
                }
            },
            "managed_objects": [],
            "optional": false,
            "prefix": null,
            "relationships": []
        }
    },
    "managed_objects": [],
    "optional": false,
    "prefix": "entry_",
    "relationships": []
}
```

3 Specification Document for brain.dataformat.brainformat

The specification document for a file format module—in this case the LBNL brain format—can be easily compiled directly from the given API modules. This allows developers to easily add new format classes and modules without having to maintain multiple documents.

```
from brain.dataformat.spec import BaseSpec
import brain.dataformat.brainformat as brainformat
json_spec = BaseSpec.compile_format_document(module_object=brainformat, as_json=True, pretty=True)

print '**' + str(time.ctime(time.time())) + '**'
print json_spec
```

```
**Sun Jul 26 22:12:21 2015**
{
    "AnnotationDataGroup": {
        "attributes": [
            {
                "attribute": "collection_description",
                "optional": false,
                "prefix": null,
                "value": null
            }
        ],
        "datasets": {
            "annotation_type_indexes": {
                "attributes": [],
                "dataset": "annotation_type_indexes",
                "description": "Dataset indicating for each selection the index of the annotation t
                "dimensions": [
                    {
                        "axis": 0,
                        "dataset": null,
                        "description": "Integer index into the annotation_types array indicating th
                        "name": "type_index",
                        "optional": false,
                        "relationships": [],
                        "unit": null
```

```
            }
        ],
        "dimensions_fixed": true,
        "optional": false,
        "prefix": null,
        "relationships": []
    },
    "annotation_types": {
        "attributes": [],
        "dataset": "annotation_types",
        "description": "List of all available annotation types",
        "dimensions": [
            {
                "axis": 0,
                "dataset": null,
                "description": "Integer index of the type",
                "name": "type_index",
                "optional": false,
                "relationships": [],
                "unit": null
            }
        ],
        "dimensions_fixed": true,
        "optional": false,
        "prefix": null,
        "relationships": []
    },
    "data_object": {
        "attributes": [],
        "dataset": "data_object",
        "description": "None",
        "dimensions": [],
        "optional": false,
        "prefix": null,
        "relationships": []
    },
    "descriptions": {
        "attributes": [],
        "dataset": "descriptions",
        "description": "Dataset with the annotation descriptions.",
        "dimensions": [
            {
                "axis": 0,
                "dataset": null,
                "description": "Integer index of the annotation",
                "name": "annotation_index",
                "optional": false,
                "relationships": [],
                "unit": null
            }
        ],
        "dimensions_fixed": true,
        "optional": false,
        "prefix": null,
        "relationships": []
    },
    "properties": {
        "attributes": [
```

```
            {
                "attribute": "name",
                "optional": false,
                "prefix": null,
                "value": null
            }
        ],
        "dataset": null,
        "description": "Datasets with a particular property for all selections.",
        "dimensions": [
            {
                "axis": 0,
                "dataset": null,
                "description": "Integer index of the selection",
                "name": "selection_index",
                "optional": false,
                "relationships": [],
                "unit": null
            }
        ],
        "dimensions_fixed": false,
        "optional": true,
        "prefix": "property_",
        "relationships": []
    },
    "selection_indexes": {
        "attributes": [],
        "dataset": "selection_indexes",
        "description": "Dataset indicating for each axis the index of the selection applied
        "dimensions": [
            {
                "axis": 0,
                "dataset": null,
                "description": "Integer index of the annotation",
                "name": "annotation_index",
                "optional": false,
                "relationships": [],
                "unit": null
            },
            {
                "axis": 1,
                "dataset": "axis_index",
                "description": "Integer index of the axis",
                "name": "axis_index",
                "optional": false,
                "relationships": [],
                "unit": "index"
            }
        ],
        "dimensions_fixed": true,
        "optional": false,
        "prefix": null,
        "relationships": []
    },
    "selections": {
        "attributes": [
            {
                "attribute": "axis",
```

```
                    "optional": false,
                    "prefix": null,
                    "value": null
                }
            ],
            "dataset": null,
            "description": "Datasets with all selections for the indicated axis. Axis -1 indica
            "dimensions": [
                {
                    "axis": 0,
                    "dataset": null,
                    "description": "Integer index of the selection",
                    "name": "selection_index",
                    "optional": false,
                    "relationships": [],
                    "unit": null
                }
            ],
            "dimensions_fixed": false,
            "optional": false,
            "prefix": "selections_axis_",
            "relationships": []
        }
    },
    "description": "Managed group for storage of a collection of annotations. Multiple annotati
    "group": null,
    "groups": {},
    "managed_objects": [],
    "optional": true,
    "prefix": "annotations_",
    "relationships": []
},
"BrainDataData": {
    "attributes": [],
    "datasets": {},
    "description": "Managed group for storage of brain data (internal and external).",
    "group": "data",
    "groups": {},
    "managed_objects": [
        {
            "format_type": "BrainDataInternalData",
            "optional": false
        },
        {
            "format_type": "BrainDataExternalData",
            "optional": false
        }
    ],
    "optional": false,
    "prefix": null,
    "relationships": []
},
"BrainDataDescriptors": {
    "attributes": [],
    "datasets": {},
    "description": "Managed group for storage of a collection of brain data descriptors.",
    "group": "descriptors",
    "groups": {},
```

```
        "managed_objects": [
            {
                "format_type": "BrainDataStaticDescriptors",
                "optional": false
            },
            {
                "format_type": "BrainDataDynamicDescriptors",
                "optional": false
            }
        ],
        "optional": false,
        "prefix": null,
        "relationships": []
    },
    "BrainDataDynamicDescriptors": {
        "attributes": [],
        "datasets": {},
        "description": "Managed group for storage of static descriptors.",
        "group": "dynamic",
        "groups": {},
        "managed_objects": [],
        "optional": false,
        "prefix": null,
        "relationships": []
    },
    "BrainDataECoG": {
        "attributes": [],
        "datasets": {
            "ecog_data": {
                "attributes": [
                    {
                        "attribute": "unit",
                        "optional": false,
                        "prefix": null,
                        "value": "Volt"
                    }
                ],
                "dataset": "raw_data",
                "description": "Dataset with the ECoG recordings data",
                "dimensions": [
                    {
                        "axis": 0,
                        "dataset": "electrode_id",
                        "description": "Id of the recording electrode",
                        "name": "space",
                        "optional": false,
                        "relationships": [],
                        "unit": "id"
                    },
                    {
                        "axis": 1,
                        "dataset": "time_axis",
                        "description": "Sample time in ms",
                        "name": "time",
                        "optional": false,
                        "relationships": [],
                        "unit": "ms"
                    },
```

```
                {
                    "axis": 0,
                    "dataset": "anatomy_name",
                    "description": "Name of region location of the electrodes",
                    "name": "space",
                    "optional": true,
                    "relationships": [],
                    "unit": "region name"
                },
                {
                    "axis": 0,
                    "dataset": "anatomy_id",
                    "description": "Integer id of the region location of the electrodes",
                    "name": "space",
                    "optional": true,
                    "relationships": [],
                    "unit": "region id"
                }
            ],
            "dimensions_fixed": true,
            "optional": false,
            "prefix": null,
            "primary": true,
            "relationships": []
        },
        "layout": {
            "attributes": [],
            "dataset": "layout",
            "description": "The physical layout of the electrodes.",
            "dimensions": [],
            "optional": true,
            "prefix": null,
            "relationships": []
        },
        "sampling_rate": {
            "attributes": [
                {
                    "attribute": "unit",
                    "optional": false,
                    "prefix": null,
                    "value": "Hz"
                }
            ],
            "dataset": "sampling_rate",
            "description": "Sampling rate in Hz",
            "dimensions": [],
            "optional": false,
            "prefix": null,
            "relationships": []
        }
    },
    "description": "Managed group for storage of raw ECoG recordings.",
    "group": null,
    "groups": {},
    "managed_objects": [
        {
            "format_type": "AnnotationDataGroup",
            "optional": true
```

```
            }
        ],
        "optional": false,
        "prefix": "ecog_data_",
        "relationships": []
    },
    "BrainDataECoGProcessed": {
        "attributes": [],
        "datasets": {
            "ecog_data": {
                "attributes": [
                    {
                        "attribute": "unit",
                        "optional": false,
                        "prefix": null,
                        "value": null
                    },
                    {
                        "attribute": "original_name",
                        "optional": true,
                        "prefix": null,
                        "value": null
                    }
                ],
                "dataset": "processed_data",
                "description": "Dataset with the ECoG recordings data",
                "dimensions": [
                    {
                        "axis": 0,
                        "dataset": "spatial_id",
                        "description": "Id of the recording electrode",
                        "name": "space",
                        "optional": false,
                        "relationships": [],
                        "unit": "id"
                    },
                    {
                        "axis": 1,
                        "dataset": "time_axis",
                        "description": "Sample time in ms",
                        "name": "time",
                        "optional": false,
                        "relationships": [],
                        "unit": "ms"
                    },
                    {
                        "axis": 0,
                        "dataset": "anatomy_name",
                        "description": "Name of region location of the electrodes",
                        "name": "space",
                        "optional": true,
                        "relationships": [],
                        "unit": "region name"
                    },
                    {
                        "axis": 0,
                        "dataset": "anatomy_id",
                        "description": "Integer id of the region location of the electrodes",
```

```
                "name": "space",
                "optional": true,
                "relationships": [],
                "unit": "region id"
            },
            {
                "axis": 2,
                "dataset": "frequency_bands",
                "description": "Frequency bands of the channels",
                "name": "channels",
                "optional": true,
                "relationships": [],
                "unit": "Hz"
            },
            {
                "axis": 2,
                "dataset": "token_id",
                "description": "Integer Id of the token type",
                "name": "channels",
                "optional": true,
                "relationships": [],
                "unit": "token id"
            },
            {
                "axis": 2,
                "dataset": "token_name",
                "description": "Name of the token type",
                "name": "channels",
                "optional": true,
                "relationships": [],
                "unit": "token name"
            }
        ],
        "dimensions_fixed": true,
        "optional": false,
        "prefix": null,
        "primary": true,
        "relationships": []
    },
    "layout": {
        "attributes": [],
        "dataset": "layout",
        "description": "The physical layout of the electrodes.",
        "dimensions": [],
        "optional": true,
        "prefix": null,
        "relationships": []
    },
    "sampling_rate": {
        "attributes": [
            {
                "attribute": "unit",
                "optional": false,
                "prefix": null,
                "value": "Hz"
            }
        ],
        "dataset": "sampling_rate",
```

```
            "description": "Sampling rate in Hz",
            "dimensions": [],
            "optional": false,
            "prefix": null,
            "relationships": []
        }
    },
    "description": "Managed group for storage of processed ECoG recordings.",
    "group": null,
    "groups": {},
    "managed_objects": [
        {
            "format_type": "AnnotationDataGroup",
            "optional": true
        }
    ],
    "optional": false,
    "prefix": "ecog_data_processed_",
    "relationships": []
},
"BrainDataExternalData": {
    "attributes": [],
    "datasets": {},
    "description": "Managed group for storage of external data related to the internal brain da
    "group": "external",
    "groups": {},
    "managed_objects": [],
    "optional": false,
    "prefix": null,
    "relationships": []
},
"BrainDataFile": {
    "attributes": [],
    "datasets": {},
    "description": "Managed BRAIN file.",
    "file_extension": ".h5",
    "file_prefix": null,
    "group": null,
    "groups": {},
    "managed_objects": [
        {
            "format_type": "BrainDataData",
            "optional": false
        },
        {
            "format_type": "BrainDataDescriptors",
            "optional": false
        }
    ],
    "optional": false,
    "prefix": "entry_",
    "relationships": []
},
"BrainDataInternalData": {
    "attributes": [],
    "datasets": {},
    "description": "Managed group for storage of a collection of internal brain data.",
    "group": "internal",
```

```
        "groups": {},
        "managed_objects": [
            {
                "format_type": "BrainDataECoG",
                "optional": true
            },
            {
                "format_type": "BrainDataECoGProcessed",
                "optional": true
            }
        ],
        "optional": false,
        "prefix": null,
        "relationships": []
    },
    "BrainDataMultiFile": {
        "attributes": [],
        "datasets": {},
        "description": "Container file used to organize multiple BrainDataFile objects into a large
        "file_extension": ".h5",
        "file_prefix": null,
        "group": null,
        "groups": {},
        "managed_objects": [
            {
                "format_type": "BrainDataFile",
                "optional": true
            }
        ],
        "optional": false,
        "prefix": null,
        "relationships": []
    },
    "BrainDataStaticDescriptors": {
        "attributes": [],
        "datasets": {},
        "description": "Managed group for storage of static descriptors.",
        "group": "static",
        "groups": {},
        "managed_objects": [],
        "optional": false,
        "prefix": null,
        "relationships": []
    },
    "ManagedObjectFile": {
        "attributes": [],
        "datasets": {},
        "description": "Container file used for external storage of managed objects. This container
        "file_extension": ".h5",
        "file_prefix": null,
        "group": "/",
        "groups": {},
        "managed_objects": [
            {
                "format_type": "ManagedObject",
                "optional": true
            }
        ],
```

```
        "optional": false,
        "prefix": null,
        "relationships": []
    }
}
```

4 License & Copyright

4.1 License

4.2 Copyright

years after the date permission to assert copyright is obtained from the U.S. Department of Energy, and subject to any subsequent five (5) year renewals, the U.S. Government is granted for itself and others acting on its behalf a paid-up, nonexclusive, irrevocable, worldwide license in the Software to reproduce, prepare derivative works, distribute copies to the public, perform publicly and display publicly, and to permit others to do so.

This page intentionally left blank.

3 SUPPLEMENT: NEUROMAP INDEX MAP RELATIONSHIP EXAMPLE

Neuromap Index Map Relationship

Oliver Rübel

June 4, 2015

1 Application Example: Using Index Map Relationship to Relate Two Images

```
In [1]: %matplotlib inline
        import matplotlib.image as mpimg
        import matplotlib.pyplot as plt
        import scipy.misc
        import numpy as np
        from tempfile import NamedTemporaryFile
        import h5py
        import sys
        sys.path.append('/Users/oruebel/Devel/BrainFormat')
        from brain.dataformat.base import RelationshipAttribute
```

1.1 Create the example image datasets

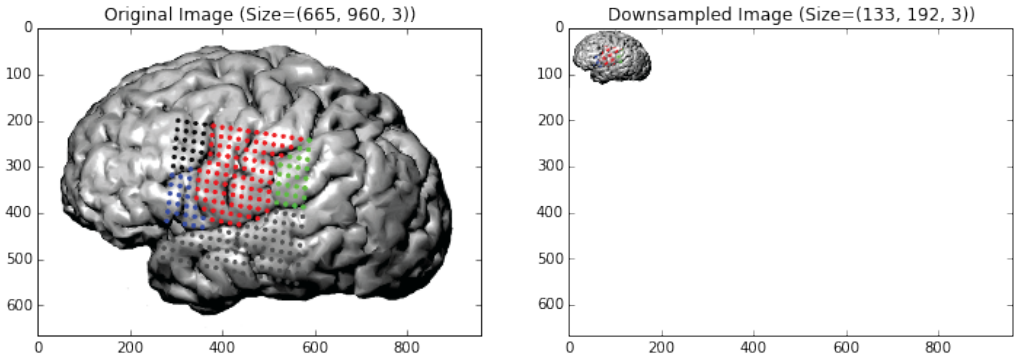
```
In [2]: # Read the image, crop large white borders, and resize the image
        # simply to speed-up the execution of this example notebook
        elec_image = scipy.misc.imresize(
            mpimg.imread('EC2.brainreg.clr.png')[20:1350, 240:2160, 0:3], 50)

        # Downsample the image to create a second dataset at 1/5th the
        # resolution using nearest-neighbor interpolation.
        elec_image_resize = scipy.misc.imresize(elec_image,
                                                size=20,
                                                interp='nearest')
```

```
In [3]: # Create the test HDF5 file
        tempfile = NamedTemporaryFile()
        test_file = h5py.File(tempfile.name, 'a')

        # Write the datasets to file
        test_file['image1'] = elec_image
        test_file['image2'] = elec_image_resize
        image1 = test_file['image1']
        image2 = test_file['image2']
```

```
In [4]: plt.rcParams['figure.figsize'] = (12.0, 10.0)
        plt.subplot(1, 2, 1)
        plt.imshow(image1[:])
        plt.title("Original Image (Size="+str(image1.shape)+")")

        plt.subplot(1, 2, 2)
```

```
plt.imshow(image2[:])
plt.title("Downsampled Image (Size="+str(image2.shape)+")")
plt.xlim(0, image1.shape[1])
plt.ylim(image1.shape[0], 0)
```

Out[4]: (665, 0)



1.2 Create the mapping between the images

Compute a map from the low-res version image2 to the full-res version image1

```
In [5]: # Initalize the array for our map
        image_map = np.zeros(shape=(elec_image_resize.shape[0],
                                    elec_image_resize.shape[1],
                                    2,     # Each index has two components (x,y)
                                    25),   # Each pixel in image2 corresponds to 25 pixel in image1
                             dtype='uint16')

        # Fill the map with the pixel correspondences
        for xi in range(image_map.shape[0]):
            for yi in range(image_map.shape[1]):
                image_map[xi, yi, 0, :] = np.resize(np.arange(xi*5, xi*5+5), 25)
                image_map[xi, yi, 1, 0:5  ] = yi*5
                image_map[xi, yi, 1, 5:10 ] = yi*5 + 1
                image_map[xi, yi, 1, 10:15] = yi*5 + 2
                image_map[xi, yi, 1, 15:20] = yi*5 + 3
                image_map[xi, yi, 1, 20:25] = yi*5 + 4
        # Save the map data to file
        test_file['image_map'] = image_map
        image_map_h5 = test_file['image_map']

        # Create optional user data documenting how the image2
        # was generated from image1
        user_description = "Resized version of target image"
        user_properties = {'algorithm': 'scipy.misc.imresize',
                           'parameters': {'interp': 'nearest',
                                          'size': 20}}
```

1.3 Create the relationship between the images

```
In [6]: # Create the index map relationship
        mapping_relationship = RelationshipAttribute.create_index_map_relationship(
            name='full resolution image',       # Name of the relationship
            map_object=image_map_h5,            # Index map
            source_object=image2,               # Source object of the relationship
            target_object=image1,               # Target object of the relationship
            map_indexing_axis=2,                # Axis in the map with index components
            map_stack_axis=3,                   # Axis in the map with the index list
            source_axis=[0,1],                  # Axes in the source to which the relationship applies
            target_axis=[0,1],                  # Axes in the target to which the relationship applies
            user_description=user_description,  # Optional user description of the relationship
            user_properties=user_properties)    # Optional user properties describing the relationship
```

1.4 Using the Relationship

Getting all relationships that define the index map relationship is simple:

```
In [7]: imr = RelationshipAttribute.get_index_map_relationship(image2, 'full resolution image')
        # Alternatively, we could have here naturally also just used the mapping_relationship
        # which were returned to us when we created the relationships,
```

Using the index map relationship we can now easily retrieve all the datasets involved in our relationships:

```
In [8]: imr_source = imr['MAP_TO_SOURCE'].target
        imr_target = imr['MAP_TO_TARGET'].target
        imr_map = imr['MAP_TO_SOURCE'].source
```

Now that we have our datasets, lets see how we can load a single pixel in our source image as well as the corresponding pixels in our target image.

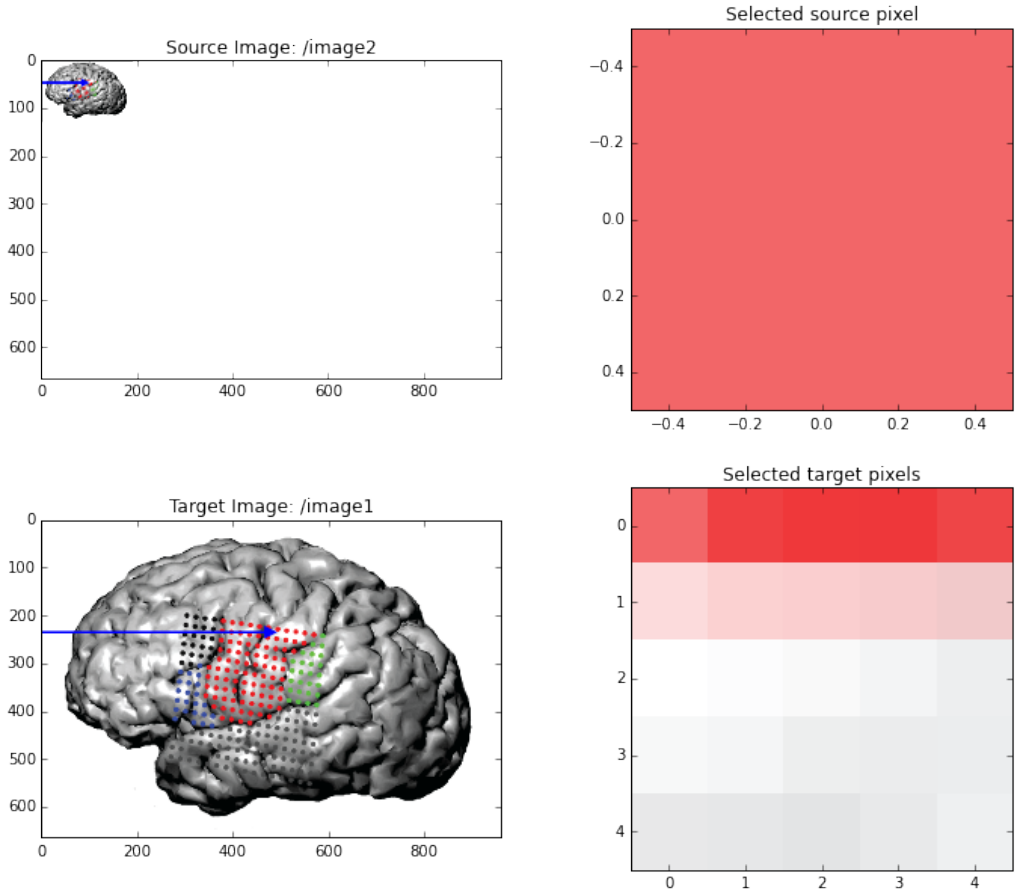
```
In [9]: # Creating a selection so that we can easily reuse it
        imr_select = np.s_[47,98]
        # Loading the selected pixel from the source (i.e., image2)
        source_data = imr_source[imr_select]
        # Mapping the selection to the target (i.e., image1)
        imr_select_target = imr['MAP_TO_TARGET'][imr_select]
        # Loading the corresponding pixels from the target dataset
        target_data = imr_target[:][imr_select_target[0,:], imr_select_target[1,:]]
```

As we can see, using relationships greatly simplifies the collaborative use of data. We were able to easily load data from our two images and we didn't even need to know the datasets nor what the mapping between our datasets was.

Now that we have loaded all our data, let's plot it to see what is going on:

```
In [10]: # Plot the source image
         plt.subplot(2, 2, 1)
         g = plt.imshow(imr_source[:])
         plt.title('Source Image: ' + imr_source.name)
         # Draw an arrow pointing to the selected pixel
         head_size = 12
         g.axes.arrow(0,
                      imr_select[0],
                      imr_select[1]-head_size,
                      0,
```

```
            head_width=head_size,
            head_length=head_size,
            color='blue')
# Set the x/y limits of the plot for direct comparison
plt.xlim(0, imr_target.shape[1])
plt.ylim(imr_target.shape[0], 0)

# Draw a plot of our selected pixel
plt.subplot(2, 2, 2)
imgplot = plt.imshow(source_data.reshape(1,1,3))
plt.title('Selected source pixel')
imgplot.set_interpolation('nearest')

# Draw a plot of our target image
plt.subplot(2, 2, 3)
g = plt.imshow(imr_target[:])
plt.title('Target Image: ' + imr_target.name)
# Draw an arrow pointing to the corresponding pixel
min_y_index = imr_select_target[1,:].min()
min_x_index = imr_select_target[0,:].min()
head_size = 22
g.axes.arrow(0,
            min_x_index,
            min_y_index-head_size,
            0,
            head_width=head_size,
            head_length=head_size,
            color='blue')

# Draw a plot of the selected target pixels
plt.subplot(2, 2, 4)
imgplot = plt.imshow(target_data.reshape(5,5,3, order='F'))
plt.title('Selected target pixels')
imgplot.set_interpolation('nearest')
```



As we can see, we selected a single pixel in the source image, which corresponds a `5x5` subimage (i.e., 25 pixels) in the target image. Since we used `nearest` neightbor interpolation when resizing the image, we see that the value of the source pixel matches the value of the top-left pixel in our target image (i.e., the selected target pixel with the smallest `x` and `y` index). The blue lines we added to the image, furthermore, confirm that we correctly selected the correct region in target based on the selection in our source.

1.5 Additional Plots for Validation

1.5.1 Specification of the Relationships that Define our Index Map Relationship

```
In [11]: print "image2 ---> order ---> image_map"
         print imr['SOURCE_TO_MAP'].relationship_spec.to_json(True)
         print ""
         print "image_map ---> indexes ---> image1"
         print imr['MAP_TO_TARGET'].relationship_spec.to_json(True)
         print ""
         print "image_map ---> order ---> image2"
         print imr['SOURCE_TO_MAP'].relationship_spec.to_json(True)
         print ""
         print "image2 ---> user ---> image1  (optional)"
         if imr['SOURCE_TO_TARGET'] is not None:
```

```
            print imr['SOURCE_TO_TARGET'].relationship_spec.to_json(True)
        else:
            print "Not set"
```

```
image2 ---> order ---> image_map
{
    "attribute": "full resolution image_IMR_SOURCE_TO_MAP",
    "axis": [
        0,
        1
    ],
    "description": "The target of this relationship defined a mapfrom the source of this relationship t
    "optional": false,
    "prefix": null,
    "properties": null,
    "relationship_type": "order",
    "target": {
        "axis": [
            0,
            1
        ],
        "dataset": "image_map",
        "filename": null,
        "global_path": null,
        "group": null,
        "prefix": null
    }
}

image_map ---> indexes ---> image1
{
    "attribute": "full resolution image_IMR_MAP_TO_TARGET",
    "axis": {
        "INDEXING_AXIS": 2,
        "STACK_AXIS": 3
    },
    "description": "The source defines a map from /image2to the target of this relationship",
    "optional": false,
    "prefix": null,
    "properties": null,
    "relationship_type": "indexes",
    "target": {
        "axis": [
            0,
            1
        ],
        "dataset": "image1",
        "filename": null,
        "global_path": null,
        "group": null,
        "prefix": null
    }
}
```

```
image_map ---> order ---> image2
{
    "attribute": "full resolution image_IMR_SOURCE_TO_MAP",
    "axis": [
        0,
        1
    ],
    "description": "The target of this relationship defined a mapfrom the source of this relationship t
    "optional": false,
    "prefix": null,
    "properties": null,
    "relationship_type": "order",
    "target": {
        "axis": [
            0,
            1
        ],
        "dataset": "image_map",
        "filename": null,
        "global_path": null,
        "group": null,
        "prefix": null
    }
}

image2 ---> user ---> image1  (optional)
{
    "attribute": "full resolution image_IMR_SOURCE_TO_TARGET",
    "axis": [
        0,
        1
    ],
    "description": "Resized version of target image",
    "optional": false,
    "prefix": null,
    "properties": {
        "algorithm": "scipy.misc.imresize",
        "parameters": {
            "interp": "nearest",
            "size": 20
        }
    },
    "relationship_type": "user",
    "target": {
        "axis": [
            0,
            1
        ],
        "dataset": "image1",
        "filename": null,
        "global_path": null,
        "group": null,
        "prefix": null
    }
```

```
}
```

```
In [ ]:
```